Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs

Christopher Stewart, Aniket Chakrabarti
The Ohio State University

Rean Griffith
VMWare

Abstract

Internet services access networked storage many times while processing a request. Just a few slow storage accesses per request can raise response times a lot, making the whole service less usable and hurting profits. This paper presents Zoolander, a key value store that meets strict, low latency service level objectives (SLOs). Zoolander scales out using replication for predictability, an old but seldom-used approach that uses redundant accesses to mask outlier response times. Zoolander also scales out using traditional replication and partitioning. It uses an analytic model to efficiently combine these competing approaches based on systems data and workload conditions. For example, when workloads under utilize system resources, Zoolander's model often suggests replication for predictability, strengthening service levels by reducing outlier response times. When workloads use system resources heavily, causing large queuing delays, Zoolander's model suggests scaling out via traditional approaches. We used a diurnal trace to test Zoolander at scale (up to 40M accesses per hour). Zoolander reduced SLO violations by 32%.

1 Introduction

Internet services built on top of networked storage expect data accesses to complete quickly all of the time. Many companies now include latency clauses in the service level objectives (SLOs) given to storage managers. Such SLOs may read, "98% of all storage accesses should complete within 300ms provided the arrival rate is below 500 accesses per second [12, 35, 39]." When these SLOs are violated, Internet services become less usable and earn less revenue. Consider e-commerce services. SLO violations delay web page loading times. As a rule of thumb, delays exceeding 100ms decrease total revenue by 1% [30]. Such delays are costly because revenue, which covers salaries, marketing, etc., far exceeds the cost of networked storage. A 1% drop in revenue can cost more than an 11% increase in compute costs [38].

Many networked storage systems meet their SLOs by scaling out, i.e., when access rates increase, they add new nodes. The most widely used scale-out approaches partition or replicate data from old nodes to new nodes and divide storage accesses across the old and new nodes, reducing resource contention and increasing throughput [12, 15, 24]. However, background jobs, e.g., write-buffer dumps, garbage collection, and DNS timeouts, also contend for resources. These periodic events can increase access times by several orders of magnitude.

Our key-value store, called Zoolander, masks slow storage accesses via replication for predictability, a historically dumb idea whose time has come [29]. Replication for predictability scales out by copying the exact same data across multiple nodes (each node is called a duplicate), sending all read/write accesses to each duplicate, and using the first result received. Historically, this approach has been dismissed because adding a duplicate does not increase throughput. But duplicates can reduce the chances for a storage access to be delayed by a background job, shrinking heavy tails[1] Very recent work has used replication for predictability but only sparingly with ad-hoc goals [2, 9, 39]. Zoolander fully supports replication for predictability at scale.

Zoolander can also scale out by reducing the accesses per node using partitioning and traditional replication. Its policy is to selectively use replication for predictability only when it is the most efficient way to scale out (i.e., it can meet SLO using fewer nodes than the traditional approaches). Zoolander implements this policy via a biased analytic model that predicts service levels for 1) the traditional approaches under ideal conditions and 2) replication for predictability under actual conditions. Specifically, the model assumes that accesses will be evenly divided across nodes (i.e., no hot spots). As a result, the model overestimates performance for traditional approaches. In contrast, our model predicts the performance of replication for predictability precisely, using first principles and measured systems data. Despite its bias, our model provided key insights. First, replication for predictability allows us to support very strict, low latency SLOs that traditional approaches cannot attain. Second, traditional approaches provide efficient scale out when system resources are heavily loaded, but replication for predictability can be the more efficient approach when resources are well provisioned.

We implemented Zoolander as a middleware for existing key-value stores, building on prior designs for high

[1]In this paper, we use the term *heavy tailed* to describe probability distributions that are skewed relative to normal distributions. Sometimes these distributions are called fat tailed.

throughput [16, 18, 39]. Zoolander extends these systems with the following features:

1. High throughput and strong SLO for read and write accesses when clients do not share keys. Zoolander also supports shared keys but with lower throughput.

2. Low latency along the shared path to duplicates via reduced TCP handshakes and client-side callbacks.

3. Reuse of existing replicas to reduce bandwidth needs.

4. A framework for fault tolerance and online adaptation.

We used write- and read-only benchmarks to validate Zoolander's analytic model for replication for predictability under scale out. The model predicted actual service levels, i.e., the percentage of access times within SLO latency bounds, within 0.03 percentage points.Replication for predictability increased service levels significantly. On the write-only workload using 4 nodes, Zoolander achieved access times within 15ms with a 4-nines service level (99.991%). Using the same number of nodes, traditional approaches achieved a service level of only 99%—Zoolander increased service levels by 2 orders of magnitude.

We set up Zoolander on 144 EC2 units and issued up to 40M accesses per hour, nearly matching access rates seen by popular e-commerce services [4, 7, 17]. We also varied the access rate in a diurnal pattern [34]. By using both replication for predictability and traditional approaches, Zoolander provided new, cost effective ways to scale. At night time, when arrival rates drop, Zoolander decided not to turn off under used nodes. Instead, it used them to reduce costly SLO violations. Zoolander's approach reduced nightly operating costs by 21%, given cost data from [17, 38]. With better data migration, Zoolander could have reduced costs by 32%.

This paper is arranged as follows: Section 2 presents Zoolander's analytic model on SLO under replication for predictability. Section 3 describes Zoolander itself and compares achieved SLOs to model predictions. Section 4 offers model-driven insights on when to use replication for predictability. Section 5 studies Zoolander at scale on EC2. Section 7 concludes.

2 Replication for Predictability

Traditional approaches to scale out networked storage share a common goal: They try to reduce accesses per node by adding nodes. While such approaches improve throughput, there is a downside. By sending each access to only 1 node, there is a chance that accessses will be delayed by background jobs on the node [9]. Normally, background jobs do not affect access times, but when they do interfere, they can cause large slowdowns. Consider write buffer flushing in Cassandra [16]. By default,

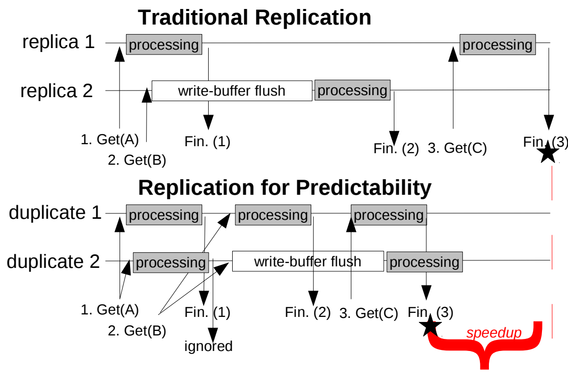


Figure 1: Replication for predictability versus traditional replication. Horizontal lines reflect each node's local time. Numbered commands reflect storage accesses. Get #3 depends on #1 and #2. Star reflects the client's perceived access time.

writes are committed to disk every 10 seconds by flushing an in-memory cache. The cache ensures that most writes proceed at full speed without incurring delay due to a disk access. However, if writes arrive randomly and buffer flushes take 50ms, we would expect buffer flushes to slow down 0.5% of write accesses ($\frac{50ms}{10s}$).

Figure 1 compares replication for predictability against traditional, divide-the-work replication. The latter processes each request on one node. When a buffer flush occurs, pending accesses must wait, possibly for a long time. However, by sending all accesses to N nodes and taking the result from the fastest, replication for predictability can mask $N-1$ slow accesses, albeit without scaling throughput. In this section, we generalize this example by modelling replication for predictability. Our analytic model outputs the expected number of storage accesses that complete within a latency bound. It allows us to compare replication for predictability to traditional approaches in terms of SLO achieved and cost.

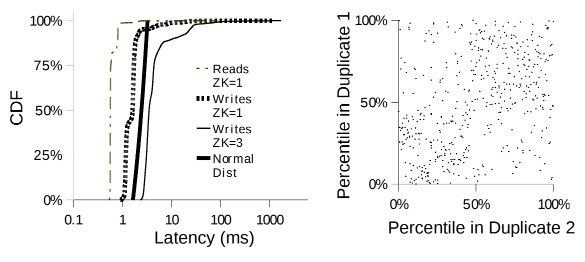
2.1 First Principles

Our model is based on the following first principles:

1. Outlier access times are heavy tailed. Background jobs can cause long delays, producing outliers that are slower and more frequent than Normal tails.

2. Outliers are non-deterministic with respect to duplicates. To mask outliers, slow accesses on 1 duplicate can not spread to others. Replication for predictability does not mask outliers caused by deterministic factors, e.g., hot spots, convoy effects, and poor workload locality.

To validate our first principles, we studied storage access times in our own local, private cloud. We use a 112 node cluster, where each node is a core with at least 2.4 GHz, 3MB L2 cache, 2GB of DRAM memory, and 100GB of secondary storage. Our virtualization software



(a) (b)

Figure 2: Validation of our first principles. (A) Access times for Zookeeper under read- and write-only workloads exhibit heavy tails. (B) Outlier accesses on one duplicate are not always outliers on the other.

is User-Mode Linux (UML) [13], a port of the Linux operating system that runs in user space of any X86 Linux system. Thus, RedHat Linux (kernel 2.6.18) serves as our VMM. Custom PERL scripts designed in the mold of Usher [26] allow us to 1) run preset virtual machines on server hardware, 2) stop virtual machines, 3) create private networks, and 4) expose public IPs. Our cloud infrastructure is compatible with any public cloud that hosts X86 Linux instances. Later in this paper, we will scale out on Amazon EC2.

We set up Zookeeper [18] and performed $100,000$ data accesses one after another. Zookeeper is a key-value store that is widely used to synchronize distributed systems. It is deployed as a cluster with ZK nodes. Writes are seen by $\frac{ZK}{2}+1$ nodes. Reads are processed by only 1 node. Figure 2(a) plots the cumulative distribution function (CDF) for Zookeeper under read-only and write-only workloads. The coefficient of variation ($\frac{\sigma}{|\mu|}$), or COV, shows the normalized variation in a distribution. Generally, COV equal or below 1 is considered low variance. We compared the plots in Figure 2(a) by 1) computing COV before the tail, i.e., up to the 70^{th} percentile and 2) computing COV across the whole CDF. Before the tail, COV was below 1. Across the entire distribution, COV was much higher, ranging from 1.5–8.

To visually highlight the heaviness of the tails, Figure 2(a) also plots a normal distribution with standard deviation and mean that were 25% larger than 90% of write times in ZK=1. Note, COV in an normal distribution is 1. The tails for both reads and writes under ZK=1 overtake the normal distribution, even though the normal distribution has a larger mean. We also found that tails became heavier as complexity increased. Writes in a single-node Zookeeper led to local disk accesses that didn't happen under reads. Writes in 3-node Zookeeper groups send network messages for consistency.

We can also interpret each (x,y) point in Figure 2(a) as a latency bound and an achieved service level. If access times followed a normal distribution, a latency bound that was 3 times the mean would provide a service level of 99.8%. Figure 2(a) shows that Zookeeper's service levels were only 98.8% of reads,96.0% of 1-node writes, and 91.5% of 3-node writes under that latency bound. To support a strict SLO that could cover 99.99% of data accesses, the latency bound would have risen to 16X, 26X, and 99X relative to the means.

Heavy tails affect many key value stores, not just Zookeeper. Internal data from Google shows that a service level of 99.9% in a default, read-only BigTable setup would require a latency bound that is 31X larger than the mean [9]. Others have noticed similar results on production systems [6, 17]. We also measured read access times in a single Memcached node, a key-value widely used in practice and in emerging sustainable systems [4, 31]. We saw a coefficient of variation of 1.9, and, under a lax latency bound, only a 98.3% service level was achieved. Finally, we ran the same test with Cassandra [16], another widely used key-value store, deployed on large EC2 instances. The coefficient of variation was 6.4.

Figure 2(b) highlights principle #2. Across two Zookeeper runs that receive the same requests under no concurrency, we show the percentile of each storage access. If slow service times were workload dependent, either the bottom right or upper left quartiles of this plot would have been empty, i.e., slow accesses on the first run would be slow again on the second. Instead, every quartile was touched.

2.2 Analytic Model

This subsection references the symbols defined in Table 1. Our model characterizes the service level provided by N independent duplicates running the exact same workload. The latency bound (τ) for the SLO is given as input. Written in plain english, *our model predicts that $\hat{s}$ percent of requests will complete within τ ms.*

$\hat{s}$	Expected service level
N	Number of duplicates used to mask anomalies
τ	Target latency bound
$\Phi_n(k)$	Percentage of service times from duplicate n with latency below k
λ	Mean interarrival rate for storage accesses
μ_{net}	Mean of network latency between duplicates and storage clients
μ_{rep}	Mean delay to duplicate a message one time plus the delay to prune a tardy reply
μ_n	Mean service time for duplicate n (derived)

Table 1: Zoolander inputs.

Using principles #1 and 2, we first model the probability that the fastest duplicate will meet an SLO latency bound. Recall, writes are sent to all duplicates, so any duplicate can process any request. Handling failures is treated as an implementation issue, not a modelling issue. The probability that the fastest duplicate responds within latency bound is computed as follows:

$$\hat{s} = \sum_{n=0}^{N-1}[\Phi_n(\tau) * \prod_{i=0}^{n-1}(1-\Phi_i(\tau))]$$

To provide intuition into this result, consider $\Phi_i(\tau)$ is the probability that duplicate i meets the τ ms latency bound. If $N=2$, $\Phi_1(\tau)*(1-\Phi_0(\tau))$ is the probability that duplicate 1 masks a SLO violation for duplicate 0. Intuitively, as we scale out in N, each term in the sum is the probability that the n^{th} duplicate is the firewall for meeting SLO, i.e., duplicates $0..(n-1)$ take too long to respond but n meets the bound. When all duplicates have the same service time distribution, we can reduce the above equation to a geometric series, shown below. (Note, as N approaches infinity, $\hat{s}$ converges to 1.)

$$\hat{s} = \sum_{n=0}\Phi_n(\tau)*(1-\Phi_n(\tau))^n = 1-(1-\Phi_n(\tau))^N$$

Queuing and Network Delay: SLOs reflect a client's perceived latency which may include processing time, queuing delay, and network latency. Since duplicates execute the same workload, they share access arrival patterns and their respective queuing delays are correlated. Similarly, networking problems can affect all duplicates. Here, we lean on prior work on queuing theory to answer two questions. First, does the expected queuing level completely inhibit replication for predictability? And second, how many duplicates are needed to overcome the effects of queuing? The key idea is to deduct the queuing delay from τ in the base model. Intuitively, requiring all duplicates to reduce their expected service time in proportion to the expected queuing delay.

$$\tau_n = \tau - (\frac{1+C_v^2}{2} * \frac{\rho}{1-\rho} * \mu_n) - \mu_{net}$$

$$\hat{s} = \sum_{n=0}^{N-1}[\Phi_n(\tau_n) * \prod_{i=0}^{n-1}(1-\Phi_i(\tau_i))]$$

We used an M/G/1 queuing model to derive the expected queuing delay, reflecting the heavy-tail service times observed in Figure 2(a). To briefly explain the first equation above, an M/G/1 models the expected queuing delay as a function of system utilization (ρ), distribution variance (C_v^2), and mean service time. Utilization is the mean arrival rate divided by the mean service time. Note, that the new τ may be different for each node (parameterizing it by n). An M/G/1 assumes that inter-arrivals are exponentially distributed. This may not be the case in all data-intensive services. A G/G/1 with some constraints on inter-arrival may be more accurate. Alternatively, an M/M/1 would have simplified our model, eliminating the need for the squared coefficient of variance (C_v^2). Prior work has shown that multi-class M/M/1 can sometimes capture the first-order effects of M/G/1.

We deduct the mean time lost to network latency. Here, network latency is the average delay to send a TCP message between any two nodes.

Multi-cast and Pruning Overhead: Replication for predictability incurs overhead when messages are repeated to all duplicates and when unused messages are pruned. These activities become more costly as the number of duplicates increase. We use a linear model to capture this. Note, we expect emerging routers to provide multi-cast support that reduces this overhead a lot. However, storage systems that use software multi-cast, like Zoolander, should consider this overhead.

$$\tau_n = \tau - (\frac{1+C_v^2}{2} * \frac{\rho}{1-\rho} * \mu_n) - \mu_{net} - N * \mu_{rep}$$

Discussion: With a nod toward systems builders, we kept the model simple and easy to understand. Most inputs come from CDF or arrival-rate data that can be collected using standard tools. The model does not capture non-linear correlations between outliers, resource dependencies, or the root causes of SLO violations.

3 Zoolander

Zoolander is middleware for existing key-value stores. It adds full read and write support for replication for predictability. Figure 3 highlights the key components of Zoolander. In the center of the figure, we show that keys are stored in *duplicates and partitions*. A duplicate abstracts an existing key-value store, e.g., Zookeeper or Cassandra. As such, a duplicate may span many nodes but it does not share resources with other duplicates.

A partition comprises 1 or more duplicates. Storage accesses are sent to all duplicates within a partition—i.e., duplicates implement replication for predictability. Storage accesses are sent to only 1 partition. There is no cross-partition communication. A global hash function maps keys to partitions. All of the keys mapped to a partition comprise a *shard*.

Zoolander can scale out by reducing storage accesses per node via partitioning. It can also scale out by adding duplicates. At the top of Figure 3, we highlight the Zoolander manager which uses our analytic model to scale out efficiently. The manager takes as input a target service level and latency bound. It also collects CDF data on service times, networking delays, and arrival rates per shard. The manager then uses our model from Section 2 to find a replication policy that meets the target SLO. It finds a policy by iteratively 1) moving a shard from one partition to another, 2) placing a shard on a new partition, and 3) adding/removing duplicates from a partition. The first and second options change the arrival rate for each

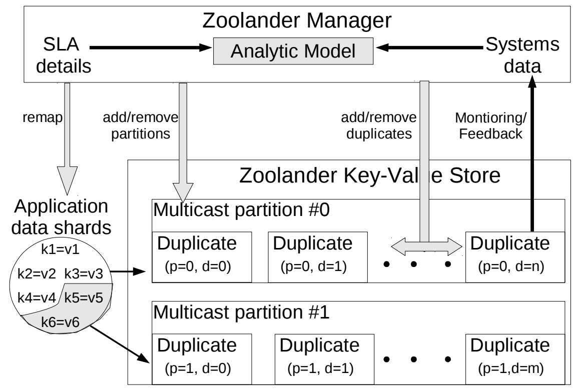


Figure 3: The Zoolander key value store. SLA details include a service level and target latency bound. Systems data samples the rate at which requests arrive for each partition, CPU usage at each node, and network delays. We use the term *replication policy* as a catch all term for shard mapping, number of partitions, and the number of duplicates in each partition.

partition and are captured by our queuing model. The third option is captured by our geometric series.

3.1 Consistency Issues

A read after write to the same key in Zoolander returns either a value that is at least as up to date as most recent write by the client (read my own write) or the value of an earlier, valid write (eventual). We can also support strong consistency funneling all accesses through a single multicast node. However, we rarely use strong consistency in any Zoolander deployments. As many prior works have noted [12, 18, 24, 39], read-my-own writes and eventual consistency normally suffice.

To support read-my-own-write consistency, each duplicate processes puts in FIFO order. Gets (reads) may be processed out of order. Clients accept reads only if the version number exceeds the version produced by their last write. For eventual consistency, Zoolander clients ignore version numbers. Figure 4 clearly depicts the supported consistency. Read my own write avoids stale data but gives up redundancy.

Propagating Writes: To ensure correct results, writes must propogate to every duplicate and every duplicate must see writes in the same order. Zoolander achieves this by using multicast. Zoolander's *client side* library keeps IP addresses for the head node of each duplicate. When client's issue a put request, the library issues D identical messages to each duplicate in a globally fixed order. In the future, we hope to replace this library with networking devices with hardware support for multicast.

Software multicast ensures that writes from a single client arrive in order, but writes from different clients can arrive out of order. We assume that multiple clients rac-

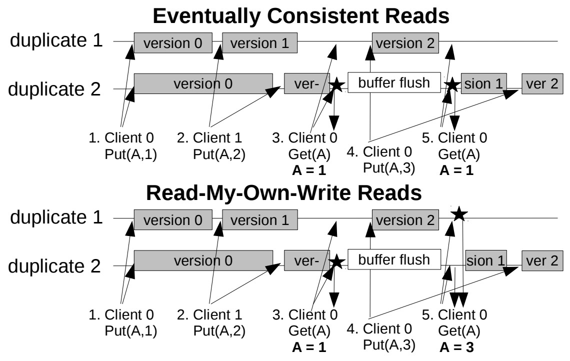


Figure 4: Version based support for read-my-own-write and eventual consistency in Zoolander. Clients funnel puts through a common multicast library to ensure write order. The star shows which duplicate satisfies a get. Gets can bypass puts.

ing to update the same key is *not* the common case. As such, Zoolander provides a simple but costly solution. To share keys, clients funnel writes through a master multicast client. This approach sacrifices throughput but ensures correct results (see Figure 4).

Choosing the Right Store: By extending existing key value stores, Zoolander inherits prior work on achieving high availability and throughput. The downside is that there are many key value stores; each tailored for high throughput under a certain workload. Zoolander leaves this choice to the storage manager. In our tests, the default store is Zookeeper [18] because of its wait-free features. However, for online services that need high throughput and rich data models [7, 8, 14], we extend Cassandra [16]. We have also run tests with in-memory stores Redis and Memcached.

3.2 Implementation Issues

Overhead: Our software multicast is on the datapath of every write; It must be fast. Our multicast library avoids TCP handshakes by maintaining long-standing TCP connections between clients and duplicates. Also, Zoolander eschews costly RPC in favor of callbacks. Clients append a writeback port and IP to every access that goes through our multicast library. Duplicates respond to clients directly, bypassing multicast. We measured the maximum number of writes, read-my-own reads, and eventual reads supported per second in Zoolander with Zookeeper as the underlying store. Table 2 compares the results to the throughput of Zookeeper by itself [18]. These tests were conducted on our private cloud.

Bandwidth: Each duplicate receives the same workload and uses the same network bandwidth. At scale, duplicates could congest datacenter networks. Zoolander takes 2 steps to use less bandwidth. First, writes return only "OK" or "FAIL", not a copy of data. Second, for

Relative Throughput & Processing Overhead		
Writes	Read-my-own-write Reads	Eventual Reads
95%(48us)	94%(52us)	99%(<1us)

Table 2: Zoolander's maximum throughput at different consistency levels relative to Zookeeper's [18]. In parenthesis, average latency for multicast and callback.

reads, Zoolander re-purposes replicas set up for fault tolerance as duplicates. Such replicas are common in production [12, 39]. Figure 5(A) compares the bandwidth used by naive support for replication for predictability against Zoolander's approach. The baseline is the bandwidth consumed by a 3-node quorum system [12, 39]. Our approach lowers bandwidth usage by 2X.

Dyanamic Systems Data: Zoolander continuously collects data using sliding windows. To keep overhead low, we collect data for only a random sample of storage accesses. For each sampled access, we collect response time, service time, accessed shard number, and network latency. A window is a fixed number of samples.

We compute the mean network latency and arrival rate for each window. We use the information gain metric to determine if our CDF data has diverged. If we detect that the CDF may have diverged, we collect samples more frequently, waiting for the information gain metric to converge on new CDF data. Figure 5 demonstrates the benefits of service time windows. First, we ran our e-science workload (Gridlab-D), then we injected an additional write-only workload on the same machine, and finally we added a read-only workload also. Our sliding windows allow us to capture accurate service time distributions shortly after each injection, as shown by convergence on information gain.

Fault Tolerance: Zoolander can tolerate duplicate, partition, software multicast, and client failures. Duplicate failures are detected via TCP Keep Alive by the software multicast. Every duplicate receives every write, so between storage accesses, software multicast can simply remove any failed duplicate from the multicast list.

A partition fails when its only working duplicates fails. When this happens, Zoolander manager uses transaction logs from the last surviving duplicate to restart the partition. This takes minutes but is automated. Software multicast is a process in the client-side library. On restart, it updates its multicast list with Zoolander manager. This process takes only milliseconds. However, when software multicast is down, the entire partition is unavailable.

3.3 Model Validation & System Results

Thus far, we have developed an analytic model for replication for predictability. We have also described the system design for Zoolander, a key value store that fully supports replication for predictability at scale. Here, we show that Zoolander achieves performance expected by our model and that the model has low prediction error.

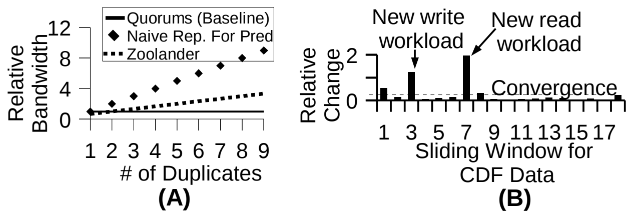
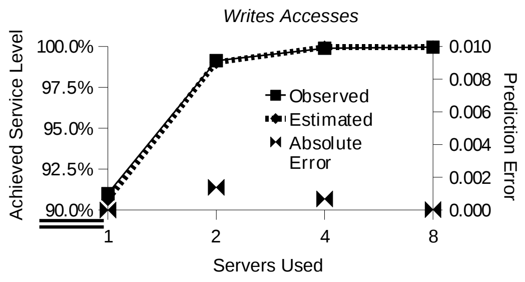


Figure 5: (A) Zoolander lowers bandwidth needs by re-purposing replicas used for fault tolerance. (B) Zoolander tracks changes in the service time CDF relative to internal systems data. Relative change is measured using information gain.

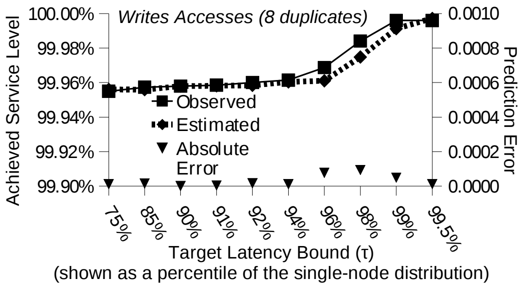
We deployed Zoolander on the private cloud described in Section 2. We used Zookeeper as the underlying key-value store. We focus on data sets that fit within memory (i.e., in-memory key-value stores backed up with local disk). We used 1 partition for these tests. We issued 1M write accesses in sequence without any concurrency. We used the 90th percentile of the collected service time distribution as the default latency bound (τ=5ms). The average response time in this setup was 3ms, so our latency bound allowed only 2ms for outliers. The SLO for Zookeeper without Zoolander was: *90% of accesses will complete within 5ms.*

We added duplicates to Zoolander one at a time, issuing the same write workload each time we scaled out. Figure 6(a) shows Zoolander's performance, i.e., achieved service level, as duplicates increase. Specifically, the achieved service level grew as duplicates were added. For example, under 8 instances, Zoolander could support the following SLO: *99.96% of write accesses will complete within 5ms*. The graph also shows that Zoolander had absolute error (i.e., actual service level minus predicted) below 0.002 in all cases. **This is a key result: Scaling out via replication for predictability strengthens SLOs without raising latency bounds.**

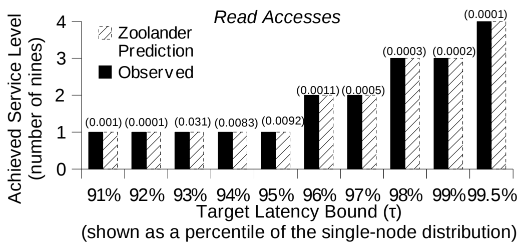
In our next test, we set the number of duplicates to 8. We used the same service time distribution from above. We then changed the latency bound (τ) to different percentiles in the single-node distribution, from the 75th to 99.5th. High percentiles led to several-nine service levels in Zoolander, forcing our model to be accurate with high precision. Low percentiles required Zoolander to accurately model more accesses. Figure 6(b) shows our model's accuracy as the latency bound increased. Absolute error was within 0.0001 for high and low percentiles. In Figure 2(a), we observed that write access times had a heavy-tail distribution that started around the 96th percentile. Figure 6(b) shows a steeper



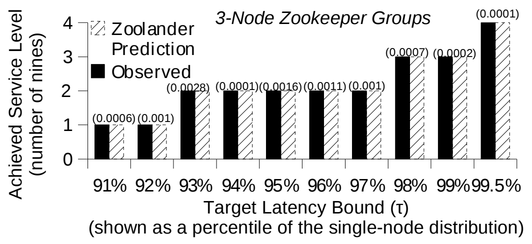
(a) Achieved service levels against Zoolander predictions as duplicates increase. Observed and estimated lines overlap.



(b) Service levels as the target latency bound changes.



(c) Service levels achieved on read-only accesses. 2 duplicates used.



(d) Service levels achieved under 3-node Zookeeper deployment. 2 duplicates used.

Figure 6: Validation Experiments

slope (strong gains) for latency bounds after the 96^{th} percentile. For instance, setting the latency bound to the 99^{th} percentile of single-node distribution (τ=15ms), RP Zookeeper achieved 99.991% service level using only 4 duplicates. In other words, adding duplicates scaled the service level by two orders of magnitude.

Diverse Workloads: Figures 6(c) and 6(d) shows the number of nines achieved under read accesses and under larger Zookeeper cluster size. On our cloud platform, reads completed in microseconds [18]. Sometimes our software repeater had not finished broadcasting accesses before a duplicate finished the job. Figure 6(c) shows the results with just 2 duplicates. As we varied the latency bound, Zoolander accurately estimated service level. We focus on the number of nines because it is a common metric in practice for SLAs. Zoolander and our model agreed on the number of nines.

Figure 6(d) shows results where we set the cluster size to 3 under a write workload. Zookeeper uses an atomic broadcast to issue cluster writes. Communication within duplicate clusters increases anomalies. Despite this increase, Zoolander met our model's expectations across all tested latency bounds.

Heterogeneous Platforms: In our toughest test for Zoolander, we made a fundamental, runtime change to our cloud platform: We allowed instances to share the L2 cache. We started Zoolander with a CDF based on private L2 caches and used our continuous monitoring to discover the new CDF (window size was 10,000). We ran a total of 1M accesses. Our input latency bound (τ) was set to the 60^{th} percentile of single-node, private-L2 service time distribution (just 3ms). A 16-instance Zoolander achieved a service level of 99.916% under this latency bound. Our full model predicted 99.927%. Table 3 shows the absolute percentage point error of different versions of the Zoolander model. The geometric series and continuous monitoring improve accuracy most.

1 node CDF	Base Geom. Series	w/ Sliding window	Full model w/ μ_{net} & μ_{rep}
60.0000	0.0835	0.0494	**0.0103**

Table 3: Percentage-point error of different versions of Zoolander's model (x100). Results for 16-node, shared L2 test.

4 Model-Driven SLO Analysis

Zoolander can scale out via replication for predictability or via partitioning. The analytic model, presented in Section 2, helps Zoolander manager choose the most efficient replication policy. The analytic model can also provide marginal analysis on the SLO achieved as key input parameters vary. Specifically, we varied the request arrival rate and used our model to predict SLO achieved. We fixed the number of nodes (4) and we fixed the systems data. We compared 3 replication policies: 1) using only replication for predictability (i.e., 1 partition with 4 duplicates), 2) using only traditional approaches (i.e., 4 partitions with 1 duplicate each), and 3) using a mixed approach (i.e., 2 partitions and 2 duplicates each). Note, our model predicts the same service levels under a k-duplicate partition with arrival rate λ as it does under N k-duplicate partitions with arrival rate $N * \lambda$, making our results relevant to larger systems.

Recall, our model is biased toward partitioning. We naively assume that each partition divides workload evenly with no internal hot spots or convoy effects. Thus, we are really comparing accurate predictions on replication for predictability to best-case predictions for partitioning. More generally, our model makes best-case predictions for any approach that reduces accesses per node by dividing work, including replication for throughput.

The results of our marginal analysis are shown in Figures 7(a–b). The y-axis in these figures is "goodput", i.e., the fraction of requests returned within SLO. The x-axis for these figures is the normalized arrival rate, i.e., the arrival rate over the maximum service rate. In queuing theory terminology, the normalized arrival rate is called system utilization. The latency bound changes across the figures. The results show arrival rates under which the studied replication heuristics excel. Specifically:

1. Zoolander's mixed approach, using both replication for predictability and partitioning, offers the best of both worlds. Replication for predictability alone increased service levels but only under low arrival rates. Partitioning alone supported high arrival rates but with low service levels. The mixed approach supported high arrival rates (>40% utilization) and achieved high SLO.

2. As the latency bound increased, replication for predictability supported higher arrival rates, and similarly, partitioning provided higher service levels.

3. Replication for predictability performs horribly under high arrival rates. Recall, all duplicates have the same queuing delay, once this delay exceeds the latency bound, replication for predictability offers no benefit. It's performance falls of a cliff.

4. Divide-the-work approaches simply can't achieve high service levels under tight latency bounds. When we set $\tau = 3.5$ms, goodput under traditional only fell below 94%. A mixed approach achieved 99% goodput.

Cost Effectiveness: SLO violations can be costly. For online e-commerce services, violations reduce sales and ad clicks. For data processing services, violations deprive business leaders of data needed to make good, profitable choices. All else being equal, reducing SLO violations means reducing costs. Replication for predictability reduces SLO violations but it uses more nodes. Nodes also cost; They use energy, their components (memory and disk) wear out, and they have management overheads. We used our model to study the cost effectiveness of using more nodes to reduce SLO violations.

We set a latency bound (τ) of 7ms and used systems data taken from our private cloud. We computed the number of SLO violations as the arrival rate changed. To provide intuition, the number of SLO violations is essentially the product of x and $(1-y)$ for (x,y) pairs in Figure 7b. The rate of violations (λ^{vio}) is shown below. F_{zk} represents our model with systems data from Zookeeper.

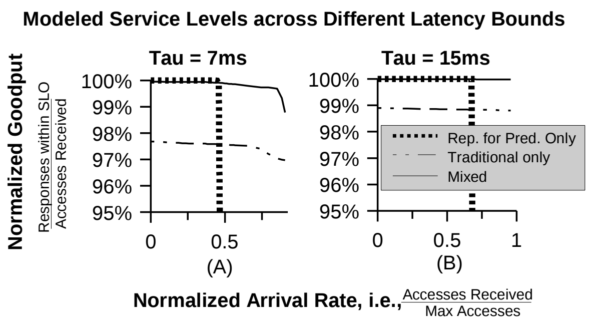


Figure 7: Trading throughput for predictability. For replication for predictability only, $\lambda = x$ and $N = 4$. For traditional, $\lambda = \frac{x}{4}$ and $N = 1$. For the mixed Zoolander approach, $\lambda = \frac{x}{2}$ and $N = 2$. Our model produced the Y-axis.

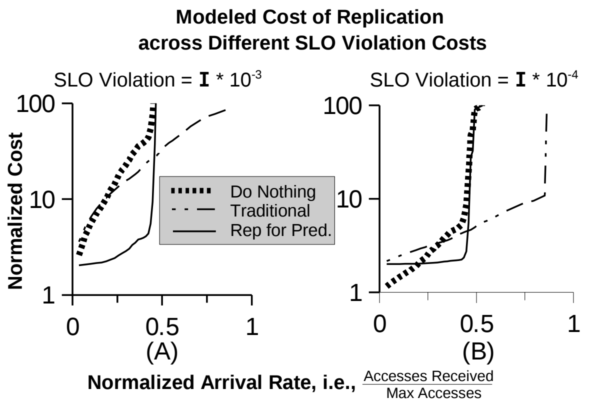


Figure 8: Cost of a 1-node system, 2 partition system, and 2 duplicate system across arrival rates. Lower numbers are better.

$$\lambda^{vio} = \lambda * (1 - F_{zk}(\tau, \lambda))$$

We used a linear model to assess cost effectiveness. Total cost was the sum of 1) SLO violations (λ^{vio}) multiplied by cost per violation (cpv) and 2) nodes used (N) multiplied by the cost per node per unit time (cpn). The model is shown below.

$$cost = N * cpn + \lambda^{vio} * cpv$$

The cost per violation and cost per node vary from service to service. We studied the relative cost between these parameters. Specifically, we set $cpn = 1$ and varied cpv, as shown in Figures 8(a-b).

We compared three replication policies. The default approach, or "do nothing", did not scale out. It used 1 1-duplicate partition ($N = 1$) and allowed SLO violations to increase with the arrival rate. The replication for predictability approach used 1 2-duplicate partition ($N = 2$) and reduced SLO violations under low arrival rates. The traditional approach used 2 1-duplicate partitions ($N = 2$). Note, N refers to the number of duplicates—each duplicate could comprise many nodes. We found the following insights:

1. When 100 SLO violations cost more than a node, replication for predictability is cost effective, until queuing delay exceeds the latency bound and service levels fall of the cliff.

2. If SLO violations are cheap, e.g., a node costs more than 10,000 violations, replication for predictability is never cost effective, even under low arrival rates.

3. If arrival rates change, the most cost effective approach will also change. When SLO violations are neither cheap nor expensive, all three approaches can be cost effective under certain rates.

The exact cost of an SLO violation depends on the service. Online services have found ways to compute *cpv* for their workloads. It is harder to compute *cpv* in emerging services, e.g., Twitter trend analysis or smart-grid power management. In these services, violations map only indirectly to revenue. However, if such violations lead to stale results that lead to poor decisions, the real cost of such violations can be very high.

5 Zoolander in Action

For this section, we studied Zoolander under intense arrival rates, e.g., workloads produced by online services. These tests used up to 144 Amazon EC2 units. EC2 is widely used by e-commerce sites and web portals. It's prices are well known. Our goal was to compare Zoolander scaling strategies and to highlight real world settings where replication for predictability is cost effective.

Many online services see diurnal patterns in the arrival rates of user requests [4, 34]. Request arrival rates can fall by 50% between 12am–4am compared to daily peaks between 9am–7pm. As a result, fewer nodes are needed in the night than in the day time. Nonetheless, services must buy enough nodes to provide low response times under peak arrival rates. Some services save energy by using only a fraction of their nodes during the night, turning off unused nodes. However, in datacenters, energy costs are low at night (because demand for electricity is low). Nighttime energy prices below \$0.03 are common. The typical service would save only \$0.12 per night by turning off a 1KW server during this period. Zoolander can exploit underused nodes in a different way; Turning them into duplicates to reduce SLO violations.

We compared the opportunity costs of reducing SLO violations against turning off machines. Figure 9 shows the competing replication policies. During the daytime, each node is needed for high throughput and operates under its max arrival rate. However, at nighttime, the arrival rate drops by 50%, allowing us to place 2 shards on 1 node or to use replication for predictability. To save energy at night, our replication policy consolidates shards, using as few nodes as possible without exceeding the peak per-node arrival rate. Our workload accesses all shards evenly, i.e., no hot spots.

Replication for predictability can be applied naively on top the energy saving approach by using idle nodes as duplicates. SCADS manager adopted this approach [39]. However, our findings in Section 4 suggest that arrival rates on nodes that use replication for predictability should be low. We decided to use replication for predictability more sparingly, keeping arrival rates low for the duplicates. For every 6 nodes, we placed 4 shards on 2 nodes (like in the energy saving approach). The remaining 4 nodes hosted 2 shards via 2 2-duplicate partitions. Our approach had 9.7% fewer violations compared to the naive approach described above.

To make the test realistic, we setup Zoolander on EC2 and tried to mimic the scale of TripAdvisor's workload. Public data [14] shows that TripAdvisor receives 200M user requests per day. On average, each user request accesses the back-end Memcached store 7 times, translating to 1.4B storage accesses per day. Learning from recent studies, we assumed the arrival rate would drop by 50% [4]. Our goal was to support 29M accesses per hour.

We used 48 clients that issued a mix of 15% Gets and 85% Puts across 96 shards. Gets/Puts were issued in batches of 20, reflecting correlated storage accesses within user requests. Each batch arrived independently, leading to exponentially distributed inter-arrival times. Note, our clients followed a realistic, open-loop workload model. Duplicates in these tests were 1-node Cassandra [16]. In a 4 hour test, our clients issued over 160M key-value lookups (40M per hour).

During our tests, Zoolander achieved high throughput and fault tolerance. While these metrics do not reflect Zoolander's contribution, they are not weak spots either! To support 40M lookups per hour, Zoolander used 48 EC2 compute units with Cassandra as the underlying key-value store. Peak throughput was 431 lookups per second per EC2 unit, about 20% higher than the average achieved by Netflix operators [7]. We encountered 546 whole partition failures across the 144 nodes where either Cassandra or the software multicast crashed. During those failures, 2,929 lookups failed. Multicast, duplicates or callbacks caused 1,200 of those failed lookups.

SLO violations also occur when Zoolander migrated data to its nighttime setup. Migrations periods are shown in Figure 10(a). The figure is based on a trace from [34]. Moving to from the daytime setup to Zoolander's night-

node id	daytime setup	night time energy saver	night time Zoolander
0	accesses/hr = 1.6M hosted shard(s) = A	accesses/hr = 1.6M hosted shard(s) = A,B	accesses/hr = 1.6M hosted shard(s) = A,B
1	accesses/hr = 1.6M hosted shard(s) = B	accesses/hr = 1.6M hosted shard(s) = C,D	accesses/hr = 1.6M hosted shard(s) = C,D
2	accesses/hr = 1.6M hosted shard(s) = C	accesses/hr = 1.6M hosted shard(s) = E,F	accesses/hr = 0.8M hosted shard(s) = E
3	accesses/hr = 1.6M hosted shard(s) = D	NOT USED	accesses/hr = 0.8M hosted shard(s) = E
4	accesses/hr = 1.6M hosted shard(s) = E	NOT USED	accesses/hr = 0.8M hosted shard(s) = F
5	accesses/hr = 1.6M hosted shard(s) = F	NOT USED	accesses/hr = 0.8M hosted shard(s) = F

Figure 9: Replication strategies during the nighttime workload for an e-commerce service.

time setup needed 4 shard migrations(see Figure 9). Moving back to daytime setup needed 2 more migrations. Zoolander used existing techniques for shard migration. We measured migration-induced violations under load and added these to Zoolander's costs.

We used the cost model in Section 4 to compare the nighttime replication policies in Figure 9. We studied two different cost per node settings. In the *private cloud* setting, the cost of a node is a function of its energy usage only. We assumed a cost of \$0.03KWh and that each node (an EC2 unit) used 100W, thus $cpn = \$0.003$. In the public cloud setting, the cost of a node includes everything provided by EC2 (i.e., high availability, EBS, etc). As of this writing, *cpn* of a small EC2 unit was \$0.085 (20X more than energy only costs). The energy saving approach used 3 nodes, whereas the Zoolander approach used 6. We set the SLO latency bound (τ) for a batch of lookups to 150ms. Out of 160M requests the Zoolander approach incurred only 57K SLO violations compared to 85K in the energy saving approach—a reduction of 32%.

Figure 10(b) plots relative cost as a function of cost per 1000 violations (*cpv*). Lower numbers are better for Zoolander. The x-axis is log scale base 10. As SLO costs increase, the Zoolander approach becomes more cost effective. Without considering migration costs, the relative cost converges to 68% quickly in the private cloud setting where *cpn* is very low. Migration costs increase relative cost by 16%, but Zoolander remains highly cost effective in private clouds. Figure 10(b) shows that Zoolander spends \$0.79 to every dollar spent by energy saver approach, saving 21%. The public cloud setting requires higher *cpv* to be cost effective.

In their fiscal statement for the 4th quarter of 2011, TripAdvisor earned \$122M from click- and display-based advertising. We divided this number by 200M daily user requests to get revenue per 1,000 page views of \$6.81. Using prior research, we estimated that each SLO violation (a 100ms delay) would lead to a 1% loss in profits [30], meaning $cpv = \$0.068$. Under this setting, the Zoolander approach was cost effective for private settings and broke even with the energy savings approach under public cloud settings. When we consider migration costs for the energy savings approach, Zoolander is cost effective even for public clouds.

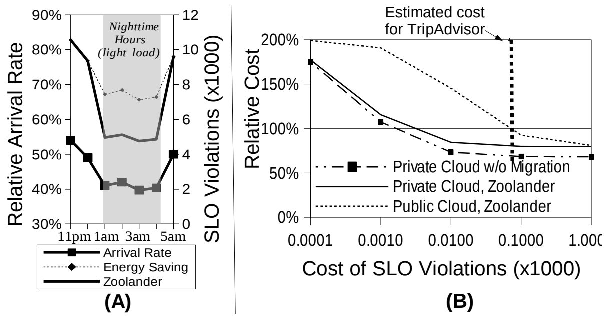


Figure 10: (A) Zoolander reduced violations at night. From 12am-1am and 4am-5am, Zoolander migrated data. We measured SLO violations incurred during migration. (B) Zoolander's approach was cost effective for private and public clouds. Relative cost is ($\frac{\text{zoolander}}{\text{energy saver}}$)

Model-Driven Management The nighttime policy for the EC2 tests was a heuristic based on insights from Section 4. Heuristics derived from principled models underlie many real world systems. Alternatively, Zoolander's model can be queried directly to find good policies.

We used systems data from Zookeeper and set τ to 3.5ms, a very low latency bound. We studied the hourly arrival rates (λ) shown in Table 4. For each rate, our model computed the expected SLO under 8 policies: 8 partitions(p) each with 1 duplicate(d), 4p with 2d, 2p with 4d, 6p with 1d, 3p with 2d, 2p with 3d, 4p with 1d, and 2p with 2d. Table 4 shows the policy that met SLO using the fewest nodes. The 5 policies shown all differ, including policies with more than 2 duplicates.

Target SLO:	*98% of accesses complete in 3.5ms*				
Accesses/Hour:	2K	850K	1M	1.5M	1.9M
Best Policy:	4p/1d	2p/2d	2p/3d	3p/2d	4p/2d

Table 4: Best replication policy by arrival rate

6 Related Work

Zoolander improves response times for key value stores by masking outlier access times. Contributions include: 1) a model of replication for predictability that is blended with queuing theory, 2) full, read-and-write support for replication for predictability, and 3) experimental results that show the model's accuracy and cost effective application of replication for predictability. Related work falls into the categories outlined below.

Replication for predictability and cloning: Google's BigTable re-issues storage accesses whenever an initial access times out (e.g., over 10ms) [9, 10]. Outliers will rarely incur more than 2 timeouts. This approach applies replication for predictability only on known outliers, reducing its overhead compared to Zoolander. Writes present a challenge for BigTable's approach. If writes that are not outliers are sent to only 1 node, duplicates diverge. If instead, they are sent to all nodes re-issued accesses would not mask delays because they would depend on slow nodes. Zoolander avoids these problems by sending all writes to all replicas.

SCADS revived replication for predictability, noting its benefits for social computing workloads [3]. SCADS sent every read to 2 duplicates [39] and supported read-only or inconsistent workloads. Replication for predictability strengthened service levels by 3–18%. Zoolander extends SCADS by scaling replication for predictability, modelling it, and supporting consistent writes. Section 5 showed that, as arrival rates increase, our model can find replication policies that outperform the fixed 2-duplicate approach.

Data-intensive processing uses cloning to mask outlier tasks. Early Map-Reduce systems cloned tasks when processors idled at the end of a job [11]. Mantri et al. [2] used cloning throughout the life of a job to guard against failures. In both cases, the number of duplicates were limited. Also, map tasks issue only read accesses. Recent work used cloning to mask delays caused by outlier map tasks [1], providing a topology-aware adaptive approach to save network bandwidth. Like Zoolander, this work focused on cost effective cloning. Zoolander's model advances this work, allowing managers to understand the effect of budget policies in advance. Another recent work [21] sped up data-intensive computing via replication for predictability. This work defines budgets in terms of reserve capacity and uses recent models on map-reduce performance [41].

Adaptive partitioning and load balancing: Heavy tail access frequencies also degrade SLOs. Hot Spots are shards that are accessed much more often than typical (median) shards. Queuing delays caused by hot spots can cause SLO violations. Further, hot spots may shift between shards over time. SCADS [39] threw hardware at the problem by migrating the hottest keys within a shard via partitioning and replication. Other works have extended this approach to handled differentiated levels of service [33] and also for disk based systems [27]. Consistent hashing provides probabilistic guarantees on avoiding hot spots [36, 42]. [19] extends consistent hashing by wisely placing data for low cost migration in the event that a hot spot arises. Locality aware placement can also reduce the impact of hot spots [23].

Both replication for predictability and power-of-two load balancing [28] involve sending redundant messages to nodes. However, in load balancing, the nodes do not share a consistent view of data. Just-idle-queue load balancing includes a related sub problem where an idle node must update exactly 1 of many queues [25]. Here, taking the smallest queue is like taking the fastest response in replication for predictability and reduces heavy tails.

Removing performance anomalies: Background jobs are not the only root cause of heavy tails, performance bugs that manifest under rare runtime conditions also degrade response times. Removing performance bugs requires tedious and persistent effort. Recent research has tried to automate the process. Shen et al. use "reference executions" to find low level metrics affected by bug manifestations, e.g., system call frequency or pthread events [32]. These metrics uncovered bugs in the Linux kernel. Attariyan et al. used dynamic instrumentation to find bugs whose manifestation depended on configuration files [5]. Recent works have found bugs at the application level [22, 40]. Debugging performance bugs and masking their effects, as Zoolander does, are both valuable approaches to make systems more predictable, but neither is sufficient. Some root causes, like cache misses [4], should be debugged. Whereas, other root causes manifest sporadically but, if they were fixed, could unmask bigger problems [35].

The operating system and its scheduler are a major reason for heavy tails. Two recent studies reworked memcached, removing the operating system from the datapath via RDMA [20, 37]. While many companies can not run applications like memcached outside of kernel protection, these studies suggest that the OS should be redesigned to reduce access-time tails.

7 Conclusion

This paper presented Zoolander, middleware that fully supports replication for predictability on existing key value stores. Replication for predictability redundantly sends each storage access to multiple nodes. By doing so, it sacrifices throughput to make response times more predictable. Our analytic model explained the conditions where replication for predictability outperforms traditional, divide-the-work approaches. It also provided accurate predictions that could be queried to find good replication policies. We tested Zoolander with Zookeeper and Cassandra. Its overhead was low. Our largest test (spanning 144 EC2 compute units) showed that Zoolander achieved high throughput and strengthened SLOs. By wisely mixing scale-out approaches, Zoolander reduced operating costs by 21%.

Acknowledgements: Thanks to Arif Merchant, John Douceur, Jian Yin, Kai Shen, and Daiyi Yang for early feedback. Nan Deng and Jaimie Kelley provided valuable feedback on Zoolander's interface. This work was funded by NSF EAGER 1230776.

References

[1] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *USENIX NSDI*, 2013.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *USENIX OSDI*, 2010.

[3] M. Armbrust, A. Fox, D. Patterson, N. Lanham, H. Oh, B. Trushkowsky, and J. Trutna. Scads: Scale-independent storage for social computing applications. 2009.

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

[5] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *USENIX OSDI*, 2012.

[6] P. Bailis. Doing redundant work to speed up distributed queries. `http://www.bailis.org/blog/`.

[7] A. Cockcroft and D. Sheahan. Benchmarking cassandra scalability on aws - over a million writes per second. `http://techblog.netflix.com`, Nov. 2011.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SOCC*, 2010.

[9] J. Dean. Achieving rapid response times in large online services, 2012.

[10] J. Dean and L. Barroso. The tail at scale. 2013.

[11] J. Dean and S. Gemawat. Mapreduce: simplified data processing on large clusters. In *USENIX OSDI*, Dec. 2004.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazons highly available key-value store. In *ACM SOSP*, 2007.

[13] J. Dike. User-mode linux.

[14] A. Gelfond. Tripadvisor architecture - 40m visitors, 200m dynamic page views, 30tb data. `http://highscalability.com`, June 2011.

[15] J. Gray. *Transaction Processing: Concepts and Techniques*. 1993.

[16] E. Hewitt. Cassandra: The definitive guide, 2011.

[17] S. Hsiao, L. Massa, V. Luu, and A. Gelfond. An epic tripadvisor update: Why not run on the cloud? the grand experiment. `http://highscalability.com/blog/2012/10/2/`.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX*, 2010.

[19] J. Hwang and T. Wood. Adaptive performance-aware distributed memory caching. In *IEEE ICAC*, 2013.

[20] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *ACM SOCC*, 2012.

[21] J. Kelley and C. Stewart. Balanced and predictable networked storage. In *International Workshop on Data Center Performance*, 2013.

[22] M. Kim, R. Sumbaly, and S. Shah. Root cause detection in a service-oriented architecture. 2013.

[23] M. Kozuch, M. Ryan, R. Gass, S. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. Lpez, M. Stroucken, and G. R. Ganger. Tashi: Location-aware cluster management. In *First Workshop on Automated Control for Datacenters and Clouds (ACDC'09)*, 2009.

[24] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *ACM SOSP*, Cascais, Portugal, Oct. 2011.

[25] Y. Lu, Q. Xie, G. Kilot, A. Geller, J. Larus, and A. Greenburg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. In *PERFORMANCE*, 2011.

[26] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. In *Proceedings of the 21st Large Installation System Administration Conference (LISA)*, November 2007.

[27] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: quality-of-service in large disk arrays. In *IEEE ICAC*, 2011.

[28] M. mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 2001.

[29] J. C. Mogul. Tcp offload is a dumb idea whose time has come. In *HotOS*, 2003.

[30] rigor.com. Why performance matters to your bottom line. `http://rigor.com/2012/09/roi-of-web-performance-infographic`.

[31] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: managing server clusters on intermittent power. In *ACM ASPLOS*, Mar. 2011.

[32] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS*, 2009.

[33] D. Shue, M. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *USENIX OSDI*, 2012.

[34] C. Stewart, T. Kelly, and A. Zhang. Exploiting non-stationarity for performance prediction. In *EuroSys Conf.*, Mar. 2007.

[35] C. Stewart, K. Shen, A. Iyengar, and J. Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *IEEE MASCOTS*, 2010.

[36] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Trans. Netw.*, 2003.

[37] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy nodes with 10gbe: leveraging one-sided operations in soft-rdma to boost memcached. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012.

[38] TripAdvisor Inc. Tripadvisor reports fourth quarter and full year 2011 financial results, Feb. 2012.

[39] B. Trushkowsky, P. Bodk, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *USENIX FAST*, 2011.

[40] W. Yoo, K. Larson, L. Baugh, S. Kim, and R. Campbell. Adp: Automated diagnosis of performance pathologies using hardware events. In *ACM SIGMETRICS*, 2012.

[41] Z. Zhang, L. Cherkasova, A. Verma, and B. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *IEEE ICAC*, Sept. 2012.

[42] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving cash by using less cache. In *USENIX Workshop on Hot Topics in Cloud Computing*, 2012.