Exoskeleton: Fast Cache-enabled Load Balancing for Key-Value Stores

Raghav Sethi

Master's Thesis

Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Master of Science in Engineering

Recommended for Acceptance
by the Department of
Computer Science

Adviser: Michael J. Freedman

June 2015

Abstract

Key-value stores are a fundamental building block for many large web-scale applications. These applications typically face highly-skewed traffic, i.e. the most popular keys are requested many orders of magnitude more frequently than the median key. Schemes that simply balance storage capacity across a cluster are therefore unable to adequately balance traffic load across cluster nodes. Dynamic load-balancing is therefore critical to maintaining service-level objectives, and necessitates expensive over-provisioning if it is imperfect or slow to deal with rapidly changing traffic distributions. An effective technique to provide load-balancing is to use popularity based caching. However, existing cache architectures have limitations - querying the cache before querying the persistent store requires a round-trip to the cache for every request, and putting a cache on the data path limits the throughput of the system to that of the cache.

Exoskeleton is a cache architecture which enables the routing of requests for popular keys directly to cache nodes at line-rate. This architecture leverages the power and speed of programmable switches to perform content-aware routing. Clients do not need to maintain any state to take advantage of this architecture, and simply encode key information directly into request packet headers. Exoskeleton removes the constraints of previous architectures, which required operators to optimize for either low latency or scalability, but not both. In addition, Exoskeleton deals with rapidly-changing workloads through a cache update mechanism designed specifically for the limitations imposed by switch hardware. Effective load-balancing enables this system to efficiently deal with modern web workloads using a heterogeneous cluster architecture comprising cheap SSD-based backend nodes and a small number of powerful cache nodes.

Acknowledgements

I would like to extend my heartfelt thanks to Mike Freedman for being an inspirational advisor, for giving me the opportunity to do work that interested me deeply, and for the time he was able to spend helping and guiding me. I especially appreciate that he did all this while he was on sabbatical and also had, as Xiaozhou said, 'two babies and a startup' to deal with.

I am grateful to my collaborators on this project - Xiaozhou Li, Mike Freedman, David Anderson (at CMU) and Michael Kaminsky (at Intel Labs). I'm indebted to Xiaozhou - without him, it would have been impossible to enjoy or even undertake a project of this scope. It has been an absolute honor to work with people I consider to be amongst the very best in the field.

I'd also like to thank my wonderful friends for keeping me sane, constantly challenging me to grow, and for making my time at Princeton one that I will always fondly remember. I hope that we still talk about the most ridiculously inconsequential things when we're old and grey. I owe much to my professors and friends at IIIT-D for believing in me and supporting me throughout my undergraduate career.

Most importantly, I would like to thank my parents and brother for being literally the most amazing people I've ever known. I cannot adequately express how much I love them, and I couldn't have done it without their unshakeable belief in me.

This work was supported by funding from Intel.

To my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The simple semantics of key-value stores allow system designers to build high-performance systems that can scale to hundreds or thousands of nodes, which is critical to delivering the throughput demanded by web-scale applications. A great deal of work has been done by the research community and the industry in designing and implementing scalable key-value systems, each with different characteristics in terms of throughput, availability, consistency, latency, fault-tolerance and persistence. Key-value systems usually scale-out by partitioning and/or replicating data across cluster nodes.

Traffic analysis of real-world key-value stores shows that the key popularity commonly follows a Zipf distribution [6]. Techniques that simply partition key space such as striping or consistent hashing are not designed to deal with highly skewed distributions. Adding replication only partially addresses the issue of load-balancing, and adds consistency-related complexity. Dynamic load-balancing techniques that involve migration of data between nodes are complex to implement in terms of indexing, routing, and consistency. More importantly, they are often too slow to deal with the rapid changes in key popularity that characterize real workload traces [11]. Even if we assume that there are no failures, these rapid changes mean that nodes relying on traditional load-balancing techniques must at least temporarily deal with traffic spikes. Given these constraints, expensive over-provisioning is often necessary to meet the strict latency SLOs that are critical to a good user experience [8].

Another artifact of real-world traffic distributions is that each node in a typical homogeneous architecture must be able to serve a small number of requests for a large number of keys in the long tail, and orders of magnitude more requests for a small set of hot keys. This indicates that heterogeneous cluster architectures with different classes of nodes serving these distinct roles are likely to be more efficient. However, optimized heterogeneous architectures are more sensitive to load-imbalances, and require fast, effective load balancing to ensure they are being utilized according to their capacity.

Caching can help provide load-balancing across cluster nodes. However, typical cache architectures fall into one of two categories:

1. Systems that first request data from the cache, and then send a request to the backing store in case of a cache miss are very popular [18]. However, unless cache hit rates are high (which requires that large amounts of data be cached), a significant fraction of requests incur the additional latency of an unnecessary round-trip to the cache.

2. Systems that place the cache in the data path for all requests (read-through cache) improve the latency of cache misses. However, this architecture has a clear scalability limitation - the overall cluster throughput is bottlenecked by the cache. The use of modern backend nodes that deliver high throughput would further exacerbate these scalability issues. Crucially, a cache failure would render the entire system unavailable.

Our caching architecture removes the need to make trade-offs between latency, throughput and scalability. We use a content-aware routing scheme in which key information is encoded into request packet headers. Commodity OpenFlow-based switches then route these requests at line rate (and with no additional latency) to either cache nodes or persistent storage nodes depending on whether the requested key is present in the cache. Clients of the system are not required keep track of cached keys and requests are routed transparently. Exploiting fast hardware optimized for packet processing in this manner has several benefits, but also means that we must deal with the limited rule-space and slow update rate of switch TCAM or L2 tables. Commodity OpenFlow switches have space available for only a few hundred thousand rules each, which in our architecture also functions as the maximum cache size.

A small cache size is not a significant limitation for our use case. Fan et. al [10] prove that the size of a popularity-based cache required to achieve load-balancing has a lower bound of $O(n \log n)$, where n is the number of backend nodes. Fan et. al also use a heterogeneous cluster architecture, however, they use a read-through front-end cache, which has the significant fault-tolerance and scalability issues we described earlier. Also, their work does not deal with traffic distributions that change over time. Our goal is to build a realistic high-performance system that can deal with rapid changes in skewed real-world traffic distributions.

The key contributions of this paper are:

1. An architecture that allows us to use popularity-based caching as a load-balancing technique, without the latency, fault-tolerance and scalability problems associated with previous caching architectures.

2. An efficient cache update mechanism to deal with the limitations imposed by switches. The algorithm aims to maximize cache utilization, minimize the cache update rate, and also react quickly to rapidly changing workloads.

3. Analysis showing that this architecture enables the use of heterogeneous clusters to deal with the traffic characteristics of modern web-applications more efficiently than traditional systems.

Chapter 2

Related Work

In this section, we provide an overview of work that we drew upon when designing and evaluating Exoskeleton.

2.1 Key-value Stores

Key-value stores are considered a fundamental building block of modern web-applications. As applications demand ever greater performance, it becomes untenable to constantly upgrade servers to deal with higher load. Instead, it is more cost-effective for operators to deploy systems that can scale out to hundreds or even thousands of nodes. Many key-value stores, including Exoskeleton, offer a simple API similar to that of a map, i.e. `GET`, `PUT` and `DELETE`. Restricted functionality allows system designers to more easily build large distributed systems as compared to traditional relational databases. However, simply enumerating API calls is not sufficient to capture the semantics of a distributed key-value store. Different systems offer different characteristics to applications in terms of throughput, availability [5][9], latency [12], fault-tolerance [9], persistence [19], consistency [15] etc.

2.1.1 Persistence

Given the performance requirements, many datastores have been built to run completely in-memory [19], be in-memory and use persistent storage for failure recovery, or function as an in-memory cache to reduce load on persistent databases [18]. Exoskeleton is designed specifically to provide efficient persistent storage through an architecture that uses both flash and RAM. This design is driven by the dramatic price decreases of fast PCIe-based flash storage in recent months, and the increasing popularity of the NVMe standard. Flash-based designs are usually cheaper up-front than architectures that store all or most of their data in main memory. Also, SSDs are also far more power-efficient than large amounts of RAM and are therefore cheaper to operate as well. These changes have not gone unnoticed by the research community (or the industry) and have led to research that addresses and optimizes for the specific performance and lifetime characteristics of flash storage [20][13].

Our cache nodes are architected similarly to Lim et. al's MICA [14], which demonstrates the benefit of a holistic approach to designing partitioned in-memory key-value stores. We also use fast user-space packet processing via Intel's Data Plane Development Kit (DPDK), and exploit parallelism by partitioning data across cores. DPDK's burst packet processing allows us to deal with skewed workloads effectively. This is critical for our cache nodes, which must deal with extremely skewed workloads.

2.1.2 Consistency

Systems such as Amazon's Dynamo [9] or Facebook's Cassandra [12] only offer eventual consistency, which makes them suitable for a narrow class of applications. Other systems, such as COPS [15] and Megastore [5] deliver stronger consistency models while attempting to retain performance. Exoskeleton offers strong consistency between the cache nodes and the persistent storage nodes, but does not currently perform any additional replication for fault-tolerance or availability. Replication is supported by the architecture of the system, and would be straightforward to implement, however we choose to focus on cluster architecture for this work.

2.2 Web Workloads

Atikoglu et al's work describing an analysis of `memcached` traffic at Facebook [4] is perhaps the most detailed look yet at a real key-value store workload. Several relevant features of realistic traffic distributions are outlined in the paper. One is the distribution of key and value sizes. Although the distributions are distinct and complex, for this work we simply note that the majority of key sizes fall below 40 bytes, and a majority of value sizes fall below 600 bytes. We pick our default key and value sizes while keeping these numbers in mind.

Perhaps most importantly, however, the authors point out the extreme levels of skew in key popularity that characterize each of the workloads they analyze. For the most generally-applicable distribution the authors studied, the bottom 50% of keys ranked by popularity account for only 1% of requests, and the top 10% of keys account for 90% of requests. By fitting these numbers into a Zipf distribution [21][23], we arrive at skewness ratio α (alpha)[1] of approximately 0.995-0.998, which is similar to, but somewhat higher than the skewness ratio of 0.99 used by YCSB [7].

Both [4] and [6] address the issue of rapid changes in traffic distributions. Bodik et al. deal with this issue by examining workload spikes. They point out the critical distinction between volume spikes [3] that can be handled by adding more stateless web servers, and data spikes that cause sudden and drastic load imbalances in partitioned databases. The authors specifically attempt to characterize and model data popularity. The datasets analyzed by the authors are heavier-tailed than standard power-law distributions - the most popular items and the least popular items are each

[1]i.e. the i-th key constitutes $1/(i_\alpha H_{n,\alpha})$ of total requests, where $H_{n,\alpha} = \sum_{i=1}^{n}(1/i^\alpha)$ and n is the total number of keys.

responsible for less traffic than expected. However, the percentage of traffic to the top 10% of keys is similar to [4], at 85% or higher in most cases. Bodik et al. also hypothesize that the number of hot objects during a spike is an absolute number instead of a fraction of all objects, as user actions in spike scenarios are likely independent of the size of the data set. Our system architecture is well-suited to deal with this, as our small cache is sized independently of the total number of items stored.

2.3 Load-balancing

Work at large web-based service providers [18] shows that typical user activity results in requests to between 20 and 100 servers, and that tail latency has an adverse impact on user experience. This impact increases dramatically with the number of servers contacted [8]. Stateful partitioned distributed services like key-value stores require effective dynamic load balancing. Unless operators over-provision to deal with imbalances, overloaded nodes cause system latency and throughput variability that prevents services from meeting strict service-level objectives (SLOs).

Various replication techniques can help address load imbalance, but each has different tradeoffs. Mitzenmacher's power-of-two-choices scheme [17] greatly improves load-balance, but assumes full replication and requires up-to-date information on queue length at each replica. Schemes that involve sending requests to multiple replicas [8] either result in wasted work or are a poor fit for systems like key-value stores where network round trips dominate total processing time. Full replication also involves high space overhead.

Dynamic load balancing techniques often involve migration of data between nodes. To reduce consistency, routing and indexing complexity, such systems usually move large blocks of data (tablets, vnodes etc.) to less-loaded nodes. This makes migration expensive and slow, and often causes significant load imbalances during the migration process [11].

Caching is an effective load-balancing technique [3]. Fan et. al's work, 'Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services' [10] most directly inspired ours. The authors point out that a skew in popularity that leads to load imbalance also necessarily improves the effectiveness of caching. Their proposed solution is a popularity-based front-end cache. They show that even under adversarial workloads, the lower bound on the size of the cache required to provide load balancing is just $O(n \log n)$, where n is the number of backend nodes. However, the authors do not discuss techniques to populate the cache or keep it up-to-date in the face of changing distributions. We have discussed in the previous section the importance of reacting quickly to sudden and drastic changes in key popularity.

Fan et al. evaluate the effectiveness of their small cache using a read-through cache architecture, which has several limitations we alluded to previously. A read-through cache is one that is placed in the critical path of all requests, which means that the throughput of the system is limited to the that of the cache. This is not a significant problem for the system described, which used far slower flash storage and dual core CPUs as back-end nodes. However, it is apparent that the architecture is not amenable to easily upgrading or adding back-end nodes. Another major issue with this architecture is one of fault-tolerance. If the cache fails, the entire cluster becomes unavailable to serve requests. This problem is further exacerbated by the fact that the architecture is designed to use a single cache node.

Chapter 3

System Design

In this section, we will discuss the design of Exoskeleton, a fast caching architecture used to perform load balancing on clusters running distributed key-value stores. Modern OpenFlow switches are an integral part of the Exoskeleton architecture, and are used to perform line-rate forwarding of requests for cached keys to cache nodes, and un-cached keys to database nodes. This form of highly effective load-balancing enables the use of specialized hardware to handle the different parts of skewed web traffic distributions. Database nodes are only required to handle the long-tail of requests, and cache nodes handle the small set of popular keys. An Exoskeleton cluster consists of several dozen ***wimpy*** database nodes, which are designed to be cheap and power efficient. The cluster also contains a small number of ***beefy*** cache nodes, outfitted with powerful processors, large L3 caches and fast NICs. Figure 3 shows the architecture of an Exoskeleton cluster. We use consistent hashing to partition key-space amongst wimpy nodes.

As we have discussed previously, Exoskeleton supports `GET`, `PUT` and `DELETE` operations. We first briefly describe the infrastructure required to process these requests, and then describe how each of these operations are processed.

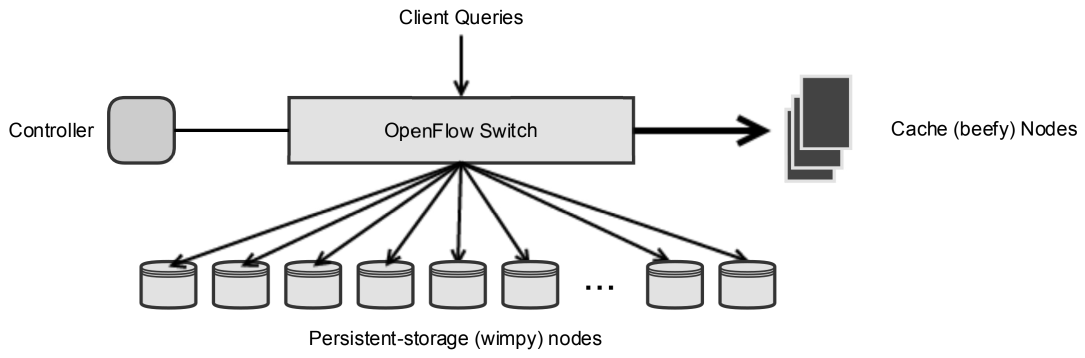


Figure 3.1: Exoskeleton Architecture

3.1 Client Library

We develop a client library that generates request packets with payloads that contain the required parameters for each request. The client library is also responsible for encoding information into packet headers such that our OpenFlow switches are able to route requests correctly. Exoskeleton uses the destination MAC address field for this purpose. The library sets the last 40 bits of the destination MAC address to a hash of the requested key. The first 8 bits are considered a prefix, and are used to identify the packet as being a request destined for Exoskeleton. Using all 48 bits may result in collisions between key hashes and actual MAC addresses of cluster node interfaces. Therefore, we select two distinct prefixes that render the resulting MAC address 'locally-administered'. This ensures that there are no collisions with physical hardware addresses. The prefixes also help the switch differentiate between GET requests (which use the first prefix) and other kinds of requests (which use the second prefix). The packet source MAC is always set to be the actual MAC of the interface used to send the packet, and is used for routing responses back to the client. Using source MACs to encode information would not pose problems for our cluster, given that we maintain control of the switch rules. However, other switches (perhaps along the path to the cluster) that interpret source MACs as indications of real nodes being connected to a specific port may cause routing issues.

We use UDP rather than TCP to reduce latency and connection-oriented overhead. Real-world workload analysis also show that most key-value pairs in practice [4] are smaller than the maximum payload size for UDP packets. Of course, if the operator controls the network equipment along the path, payloads can be much larger. Our system natively supports 8KB packets. Our architecture supports fragmenting response packets, although we do not implement this. The client library re-sends packets for which responses do not arrive within a specified timeout period, however, we observe low packet-loss rates in practice.

Clients are not required to keep track of system state at all - they do not need to know the number of nodes or their addresses, nor do they need to keep track of whether specific keys are present in the cache nodes or database nodes. The next section describes how request packets from the client are processed by the switch.

3.2 Switch Rules

Table 3.1 lists the rulesets installed into the switch in order of match priority. For clarity, the rules described in this table assume that each wimpy node is assigned a single partition of the key-space (i.e. one vnode per wimpy).

The relative priorities of these rules are critical. Ruleset 1 ensures that packets destined for specific nodes are always routed to each other. Ruleset 2 has a higher priority than Ruleset 3, so that packets that contain `GET` requests for keys that are present in the cache are first routed to the appropriate beefy node, and all remaining packets are forwarded to the appropriate wimpy node according to Ruleset 3.

Table 3.1: Rulesets installed in switch

Ruleset 1	For each node i in the set of beefy, wimpy and client nodes: {match:dest_mac=MAC$_i$ action:outport=Port$_i$} *Directs packets destined for specific nodes directly to them. Used for internal messages, and to route responses back to clients directly*
Ruleset 2	For each cache node i: For each cached key j in the cache node i: {match:dest_mac=0xAA:Hash$_j$ action:outport=Port$_i$} *Directs **GET** requests for cached items to the appropriate cache node. This ruleset must be kept up to date as the cache entries change. **0xAA** is Prefix 1*

Ruleset 3	For each partition i:
	{match:dest_mac=Prefix$_{1or2}$:{Mask$_i$} action:outport=Port$_i$}
	
	*Directs requests for items that do not match any other rules to the appropriate database node. These rules should match packets with any of the two 8-bit prefixes **0xAA** and **0xBA**.*
	
	For example, if there were only two wimpy nodes in the cluster, the two wildcard rules installed would be:
	{match:dest_mac=101*1010 ******** ******** ******** *******0 action:outport=1}
	{match:dest_mac=101*1010 ******** ******** ******** *******1 action:outport=2}

3.3 GET Requests

GET requests for a key can be received either by a beefy node (if Ruleset 2 is matched), or a wimpy node (if Ruleset 2 did not match). Requests received by beefy nodes will result in a lookup of an in-memory map, while requests received by wimpy nodes will result in a search of data present on flash storage. In case the data is found successfully, the subsequent behavior is same for both classes of nodes. They construct a response message with the key and value requested, and with the same sequence number as the incoming request. The destination MAC is set to be the source MAC of the incoming packet, which corresponds to the client's actual MAC address. The source MAC is set to be the actual MAC address of the interface on which the response is sent out. The switch will forward the response packet directly to the client by matching on Ruleset 1.

There may be small periods of time where a beefy node receives a request for a key that cannot be located in its in-memory map. This occurs due to a delay in rule removal from the switch. In this case, the beefy node needs to forward the packet to the appropriate wimpy node. However, simply sending the packet out unmodified will result in a loop, as the switch will again match on Ruleset 2. To prevent this, the beefy node changes the 8-bit prefix of the packet's destination MAC from Prefix 1 to Prefix 2 before sending the packet out. This prevents a match on Ruleset 2 (which only matches on Prefix 1), and Ruleset 3 (which matches both prefixes) will direct the packet to the appropriate wimpy node. The wimpy node will respond directly to the client as we have previously described.

3.4 `PUT` and `DELETE` Requests

Clients encode Prefix 2 into `PUT` and `DELETE` requests to ensure that requests for both cached and uncached keys are routed to database nodes rather than cache nodes. The use of Prefix 2 ensures that Ruleset 2 cannot match, and that Ruleset 3 must be used instead. This behavior is useful because there is no incremental benefit to having cache nodes process `PUT` or `DELETE` messages, as each such message must in any case be processed by the database nodes.

3.4.1 Requests for Uncached Keys

Dealing with `PUT` and `DELETE` requests destined for keys not present in the cache is straightforward. The wimpy node simply updates its persistent database, and responds to the client directly. There are no replication issues, as uncached keys have no replicas. Note that each wimpy node is required to keep track of which keys in its partitions are cached at beefy nodes so that additional measures can be taken to maintain consistency.

3.4.2 Requests for Cached Keys

`PUT` and `DELETE` requests for cached keys are more complex. For both `PUT` and `DELETE` requests, the wimpy node performs a two-phase commit. It sends a query-to-commit message to the appropriate beefy node, writes the new `PUT` value or `DELETE` marker to its log, and the old value to its undo log. The beefy node also writes the new value to its log, the old value to its undo log, and responds with an acknowledgement. On receiving this acknowledgement, the wimpy writes the new `PUT` value to the database or performs the delete, and sends a commit message to the beefy node. The beefy updates its in-memory map, and then sends back an acknowledgement message. If this is received successfully, the transaction is complete, and the wimpy sends a confirmation to the client. If not, the transaction is rolled back.

3.5 Cache Update Mechanism

Our goal is that the cache nodes store the hottest keys across the cluster at any given point in time, so as to take as much load as possible away from the resource-constrained database nodes. We now discuss our mechanism for estimating the hottest keys cluster-wide, ensuring that these keys are cached, and rapidly installing rules into the switch to reflect the contents of the cache. Figure 3.5 provides an overview of the mechanisms used to keep the cache up-to-date.

3.5.1 Load Reporting

To help determine the hottest keys across the cluster, each wimpy node sends a `REPORT` message to the appropriate beefy node at regular intervals, which contains a measure

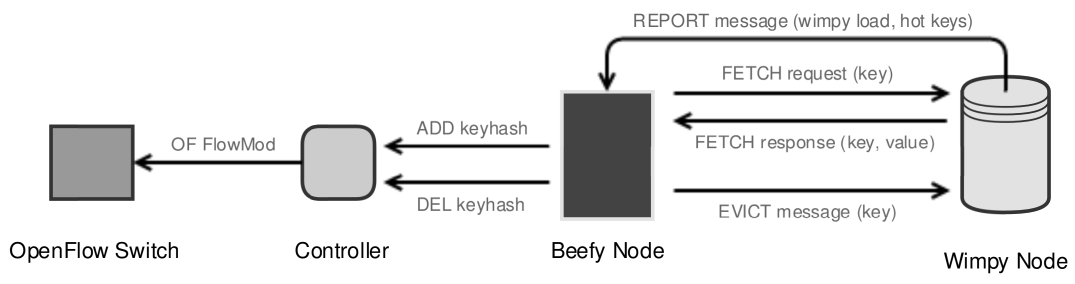


Figure 3.2: Overview of Cache Update Mechanism

of how much load it is dealing with, along with the estimated frequencies of its most popular keys. Overall wimpy load is measured by an exponential weighted moving average of its request throughput. Popular keys are tracked using a heavy-hitter estimation algorithm proposed by Metwally et al. [16]. To estimate the top-k keys in a stream, the algorithm only requires O(k) memory. This memory efficiency is critical when we consider that individual wimpy nodes can store billions of keys. However, the algorithm is too computationally expensive to run for every incoming request to a resource-constrained wimpy node. To address this, we only update the heavy-hitter data structure for a small sample of requests. As we are simply attempting to detect noticeable skew in wimpy traffic, sampling works remarkably well.

At a high level, the detection algorithm works as follows. If the request is sampled, the algorithm bumps an existing key to a bucket with a higher count, or adds a new bucket with the minimum estimated frequency for the key if less than k keys are currently tracked. If the key is untracked and k keys are already being tracked, the algorithm will evict the least recently visited key from the bucket that contains the keys with lowest estimated frequency. The new key is added and the minimum estimated frequency of the data structure is incremented.

To better take into account rapid changes in distributions, we augment Metwally's heavy-hitter detector to prioritize more recently requested keys by using individual heavy-hitter estimators for each time interval. To report the recent heavy-hitters, the database node collects keys and counts from all the time-segment counters in the list, and uses a weighted average biased towards more recent keys to estimate the current popularity of each key.

3.5.2 Selection of New Cache Entries

Database nodes send report messages at regular intervals. As discussed previously, these messages contain the estimated frequencies of the hottest keys on each wimpy node. Each beefy node maintains an ordered list of the hottest keys across all wimpy nodes assigned to it. This list is updated as new report messages arrive and are processed. At short intervals, we take a snapshot of this list. We then traverse this snapshot and add each element in the list whose load factor is greater than a threshold to the LRU map. As this is a fixed-size map, this results in the least recently used keys being evicted. We select the addition threshold to be the load of the item most-recently evicted from the LRU map. These additions and deletions to the map are also recorded as entries pushed into add and delete queues, which are used to propagate change information to the switch. The process of propagating these cache changes to the switch is described in the Section 3.6. Each addition to the cache requires that the cache node `FETCH` the values for the cached keys from the database nodes. Each cache eviction results in an `EVICT` message being generated for the appropriate database node. Actual cache additions and deletions only take place when values have been obtained from the database nodes.

3.6 Switch Update

Before we describe our update mechanism, it is important to note that the way we use switches in our architecture is quite different from their intended purpose, and this places some important restrictions on update mechanisms. We describe these limitations and present techniques we use to mitigate their effects.

The first limitation is the size of the L2 table. It is reasonable to expect OpenFlow switches to support a few hundred thousand entries in this table, although the newest switches (as of mid-2015) support as many as 1 million entries. In our architecture, this number also functions as the maximum number of cache entries supported. If our system relied on caches having high hit-ratios to deliver the required throughput, this limitation would be crippling. However, the beefy cache has a different role in our system - it exists solely to balance load across wimpy nodes. For this purpose, a small cache is completely sufficient. As Fan et al. show, the size of a popularity-based cache for load-balancing has a lower bound of only $O(n \log n)$ [10], where n is the number of database nodes, and not, as one might expect, the total number of keys stored. Therefore, hundreds of thousands of rules in the L2 table are quite sufficient to balance load amongst hundreds of database nodes.

Another significant limitation is the extremely limited rule update rate provided by typical switches. In most cache algorithms, inserting a new key into a full cache immediately triggers a cache eviction. For our system, this means for every cache update, two rule updates (delete-rule, and add-rule) must be pushed to the switch.

Consider a situation in which a rapid distribution change creates a large amount of cache churn. If our architecture required both adds and deletes to happen consecutively, we would be able to add items to our cache at only be half of our (already meager) switch update rate. This scenario would likely increase the amount of time our wimpy nodes suffer load imbalance.

Prioritizing additions over deletions would mitigate this issue. However, if we were using all of our L2 rule-space, prioritization would not help us improve our cache add rate, as we would have to necessarily delete a rule to make space for a rule that we wanted to add. To address this, we reserve a small percentage of our L2 rule space as a *fast-add buffer*. Our controller-update algorithm can use this buffer to quickly add rules without having to perform deletes immediately in case of rapid distribution changes. Temporarily delaying rule deletion has no negative impact on overall system throughput, but delayed additions of newly-popular keys could result in dropped packets or latency increases. When the cache churn rate decreases or the buffer is exhausted, the delete queue will be depleted. The size of the buffer can be tuned depending on the magnitude of the workload changes we wish to handle quickly. Algorithm 1 shows how we prioritize cache additions over deletions with a fast-add buffer.

```
while True do
    while !add_queue.empty() and !cache.full() do
        controller.push(add_queue.pop());
    end
    if !delete_queue.empty() then
        controller.push(delete_queue.pop());
    end
end
```

Algorithm 1: Cache update algorithm

It is also important to note that we use key popularity to determine the contents of the cache. LRU is quite effective at identifying the most popular keys, however, the cache churn rate is far too high for our limited rule update rate. LRU and top-k have similar hit-rates in practice, but it is clear that a list of the top-k keys in any real distribution will incur less churn than simply using a list of the most-recently accessed elements.

3.6.1 Tracking Cached Keys on Database Nodes

Wimpy nodes must keep track of which keys in their partitions are cached in a beefy node. This information is used to determine whether consistency-related operations are to be executed for an incoming `PUT` and `DELETE` requests. As `REPORT` messages only contain the most popular keys, the beefy node must retrieve the associated values to place into the cache. The beefy node selects a number of keys to cache from any given wimpy node, and sends `FETCH` requests to the wimpy nodes to obtain values for each of the keys it has decided to cache. `FETCH` responses have the same format as `GET` responses. However, `FETCH` messages also serve as a way of communicating to a wimpy node a beefy node's intention to cache a given key. Once a `FETCH` message for a specific key is received, the wimpy node will perform consistency-related operations for that key going forward.

When items are removed from the cache, the cache node sends an `EVICT` messages to the appropriate wimpy node. An `EVICT` message directs the wimpy node to remove a specific key from its list of cached keys. Once an `EVICT` message for a key is processed, `PUT` and `DELETE` messages for that key will be processed solely by the wimpy node, and no consistency-related operations are required.

3.7 Hardware Requirements

In this section, we will describe the hardware required by an Exoskeleton cluster. We utilize specific hardware functionality to build our system in a cost-effective and scalable manner.

3.7.1 Switch

A high-performance OpenFlow-enabled switch is an essential part of our architecture. A Exoskeleton-compatible OpenFlow switch is required to have a TCAM table that can store O(*num_vnodes*) (i.e. a few thousand) wildcard rules . Additionally, it must be able to hold at least O(*cache_entries*) (i.e a few hundred thousand) rules in its L2 table - which is used to perform exact matches on various L2 headers. It must also be able to set exact match rules to at a higher priority than wildcard-match rules. Some switches intelligently decide which tables rules are placed into. These switches are also compatible with our architecture, as long as our rulesets can be installed with the specified priority. In our cluster, we run a Ryu-based controller on one of the cache nodes, although this is not enforced by our architecture. The controller simply exposes a REST API that allows every cache node to send rule updates to the switch via HTTP.

3.7.2 Database (wimpy) nodes

The design of our database (*wimpy*) server was motivated by its expected workload given our two-tiered architecture. Every node must act as the backing store for billions of key-value pairs (i.e. hundreds of gigabytes) and serve as many requests as possible while remaining cheap and power-efficient. We now describe our process for selecting the appropriate hardware for our wimpy nodes.

To store hundreds of gigabytes of data, in-memory storage would be too expensive and power hungry. Assuming a majority of queries require require reads from database rather than CPU or application caches (as the wimpy is handling the long tail of requests), performance of the database is of paramount importance. Spinning-disk storage has high random seek times, and is immediately disqualified for this reason. We therefore explore various flash storage options.

As our intended application is a key-value store, we are primarily concerned with small-block random read IOPS. Given the inherently parallel architecture of flash devices, we want to maintain a high queue depth to maximize utilization. SATA is not well-optimized for parallel operations, and therefore SATA-based SSDs scale poorly with queue depth [2]. PCIe-based SSDs do not suffer from this issue. In the interests of reproducibility, we select NVMe; which is fast becoming popular as a PCIe-based standard. Multiple consumer-focused NVMe SSDs have launched in the first half of 2015. NVMe devices are also more CPU efficient (due to a lower-overhead architecture) than SATA devices.

As NVMe SSD performance scales almost linearly with queue depth, it is in our interest to select multi-core CPUs that can effectively utilize SSD I/O capacity. Batching could be used to further increase average queue depth. Inexpensive multicore SoCs have far lower clock speeds than typical high-end server chips, but given that we expect to spend a significant amount of CPU time waiting on SSD I/O we believe that the lower clock speed will help us achieve our power-efficiency goals.

3.7.3 Cache (beefy) nodes

Beefy nodes are required to deliver extremely high throughput while storing a small total amount of data from memory (hundreds of thousands of key-value pairs). We require a fast CPU for packet processing, a large L3 cache, and a fast modern NIC. Our small total cache size allows us to rely on the L3 cache to store a large portion of cache entries, rather than using expensive and power-hungry RAM. Intel Data Plane Development Kit (DPDK) [1] support for the NIC is not a requirement, but is useful for high-speed packet processing.

3.8 Software Architecture

In this section we provide a brief overview of the software architecture used for cache and database nodes, and describe the network stack for both kinds of nodes in detail.

Our cache nodes are required to comfortably support many millions of small-key value operations per second. To assist with high-speed packet processing, our cache nodes use Intel DPDK for networking rather than standard system sockets, as DPDK provides several libraries to assist with fast user-space packet processing. Exoskeleton uses the provided Poll-Mode Driver to perform low-overhead burst packet processing, avoiding expensive interrupts. We configure our NIC to improve throughput by spreading packet processing load across all available system cores. Depending on hardware capabilities, various flow/packet direction technologies can be used, including Intel Flow Director and RSS. Our architecture uses the run-to-completion model, where each core is responsible for serving a request end-to-end, as opposed to a pipeline model where one core is dedicated to reading packets and all the remaining cores performs partial packet processing.

Each cache node maintains a fixed-size in-memory map used to serve incoming requests. The map uses the LRU algorithm to evict keys that are added when the map is at capacity. The size of the map is determined by dividing the maximum number of cached items (usually capped at the size of the switch's L2 table) by the number of cache nodes. The map also contains a frequency counter for each cached item.

Given the resource constraints on our wimpy database nodes, we require low-overhead and efficient SSD-optimized database software. We use the RocksDB embedded database, as it meets all these requirements. We architect our network stack in a fashion similar to Facebook's optimized memcached distribution [22], with one thread running per-core and no direct inter-core communication. These threads each block on a shared receive socket, but possess individual send sockets. We utilize Linux raw sockets, as we want to be able to read and modify IP and Ethernet headers, and therefore can read and write entire Ethernet frames. Our implementation is designed to have as little overhead as possible. However, given that single-node key-value store performance is not the focus of this work, we refrain from modifying the kernel or RocksDB to further improve performance.

Chapter 4

Evaluation

4.1 Experimental Setup

4.1.1 Wimpy Emulator

As it was infeasible to build a large cluster of specialized wimpy nodes specifically for this project, we wrote an evaluation tool that is able to emulate multiple wimpy nodes on a powerful multicore server. We dedicate one core to each emulated wimpy which along with receive-side scaling and per-core DPDK receive queues, helps achieve a measure of isolation. The actual packet processing code for both the wimpy and emulator is precisely the same with three exceptions:

1. The emulated wimpy uses DPDK rather than raw sockets to interface with NICs and to provide burst packet I/O.

2. The emulated wimpy nodes do not actually perform disk reads - they simply return a fixed value for every requested key they receive.

3. Each emulated wimpy is rate-limited to the throughput of the actual wimpy node.

4.1.2 Test Cluster

Our test cluster contains four identical nodes in the following configuration:

- 2 x Wimpy Emulator (1-16 emulated nodes each)
- 1 x Beefy Server
- 1 x Test Client

Each server has an Intel Xeon CPU E5-2660 with 20MB of L3 cache, in a dual socket configuration with 8 physical cores per socket running at 2.20GHz. Each node is equipped with 32GB of RAM. The servers run Ubuntu 14.04, with Linux kernel 3.13. We use single-port Mellanox ConnectX-3 EN 40GbE QSFP+ PCIe-based NICs on each server. As the NIC drivers support only 16 RX/TX queues per port, we can only effectively use 8 physical (16 logical) cores per node with the DPDK run-to-completion model.

These four servers are connected via QSFP+ to a Pica8 P3922 switch with 48x10GbE ports and 4x40GbE uplink ports. The P3922 is built on a Trident+ ASIC. We run the Pica8 switch in Open vSwitch (OVS) mode, which enables OpenFlow 1.3 support. The switch supports a maximum of 128K rules in its L2 table, and 1024 rules in TCAM. As of PicOS 2.5, L2 rule update rates are limited to approximately 50 updates/sec. It is important to note that our choice of switch is also conservative in terms of OpenFlow performance. New switches from vendors like NoviFlow have 8x the rule space and 64x the update rate of the P3922.

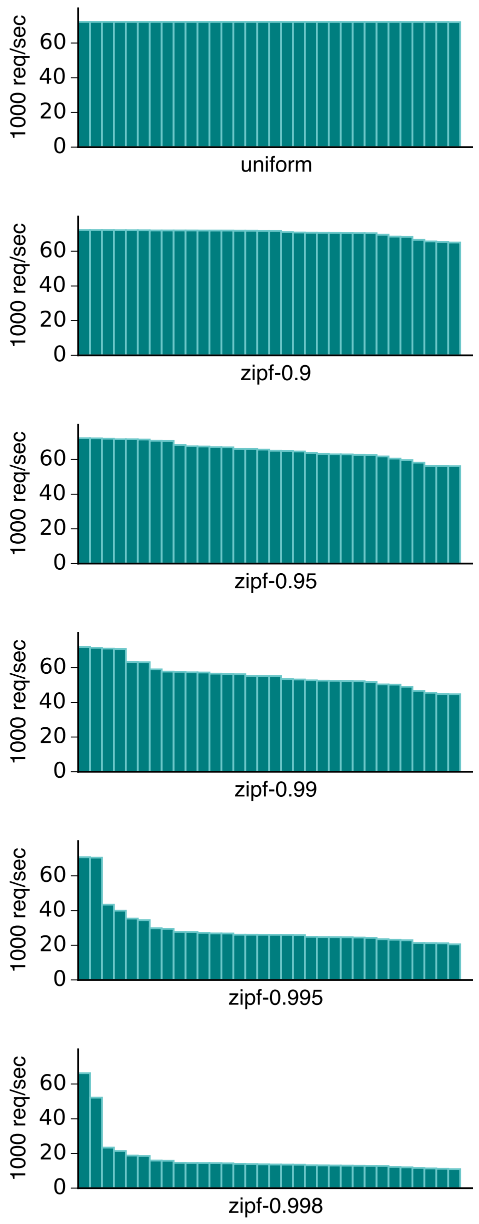
4.2 Static Distributions

Our first set of experiments is intended to evaluate the impact of a fast cache on load-balancing and system performance. In these experiments, we pre-populate our beefy caches, and install the required rules into the switch prior to starting the benchmark. We use 32 emulated wimpy nodes - 16 each on two servers. Each emulated wimpy is rate-limited to serving a maximum of 72,200 requests per second. For all experiments, keys are 16 bytes, and values are 512 bytes. Our testing shows that modifying key or value size has little impact on query throughput, as long as requests and responses can fit into a single UDP packet. We turn off reliability features for our experiments in order to avoid the high associated overhead. To make fair comparisons across different configurations, we control our client send rate to maintain a constant packet loss of approximately 5% for all experiments. The size of our total database is fixed at 100 million keys, although our architecture can easily support hundreds of billions or trillions of keys at the sizes described above.

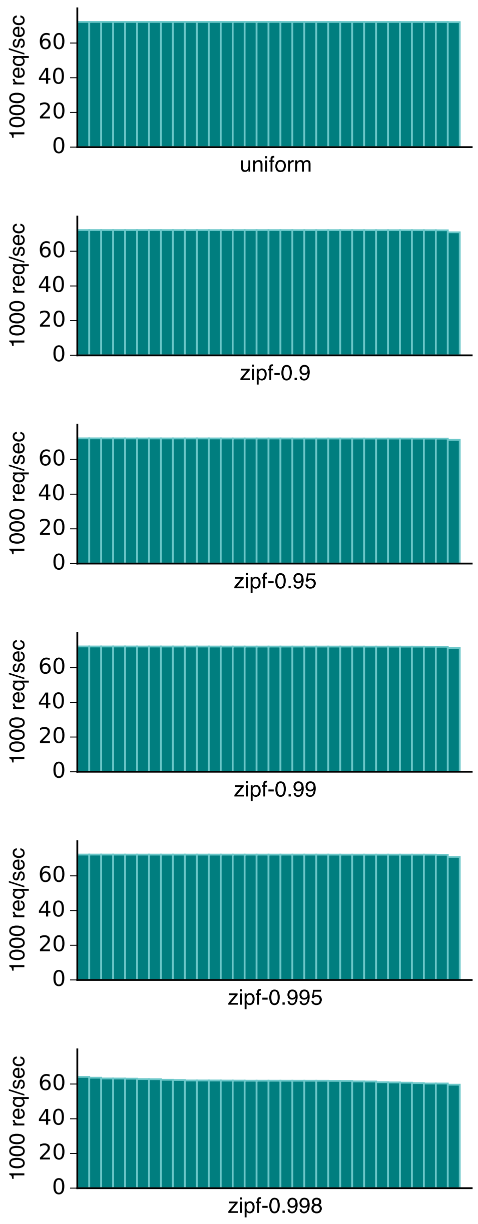
Figure 4.1 shows the impact of adding a cache with just 1,000 total entries on a cluster exposed to traffic with different levels of skew. It is clear from the Figure 4.1a that high levels of skew in un-replicated partitioned databases result in unacceptably poor average utilization resulting from significant load-imbalances. It is also apparent from the graph how effective a small, fast cache is in allowing system operators to maximize utilization for skewed distributions.

We also examine the effects of different levels of skew on the effectiveness of the cache, and its contribution to *aggregate cluster throughput.* Figure 4.2 summarizes our results. We see that at if traffic is skewed with Zipf alpha of 0.995, the addition of a cache node which serves the 1000 hottest keys increases cluster throughput by almost 5x. For higher levels of skew (>= 0.996), the single cache node handles more incoming traffic than the 32 persistent-storage nodes in the cluster.

It is also important to consider the impact of cache size on load-balancing to understand how many cache nodes and how much switch L2 rule-space is required to achieve load balancing across a given cluster. Figure 4.3 shows the impact of different cache sizes on system throughput. This experiment sets the Zipf skew to



(a) Without Cache



(b) With Cache

Figure 4.1: Throughput of wimpy database nodes for different levels of traffic skew Node IDs (x-axis) are sorted according to their throughput. A cache with just 1000 entries is extremely effective at load balancing. The more skewed distributions are also able to utilize close to 100% of wimpy node capacity.

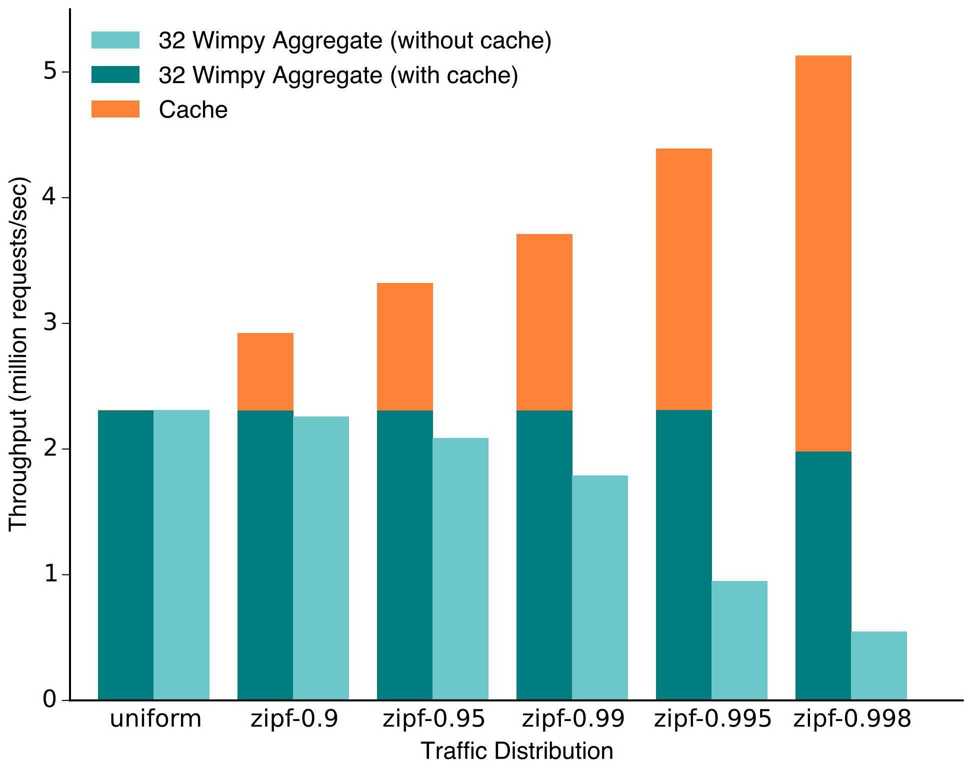


Figure 4.2: Impact of traffic skew on cluster throughput
A cache size of 1000 entries is sufficient to ensure consistent performance of the wimpy nodes whether the distribution is uniform or highly skewed.

be 0.99. As we have mentioned previously, this is a conservative estimate. Previous work[4] indicates that skews of > 0.995 are commonly observed. We see that when our cache contains 15,000 entries, our cluster throughput is below its aggregate capacity, as the percentage of traffic directed to the cache node has become so high that our wimpy nodes start to become underutilized. System operators can easily compute the appropriate cache size to achieve close to 100% utilization on every cluster node simply by examining the traffic skew.

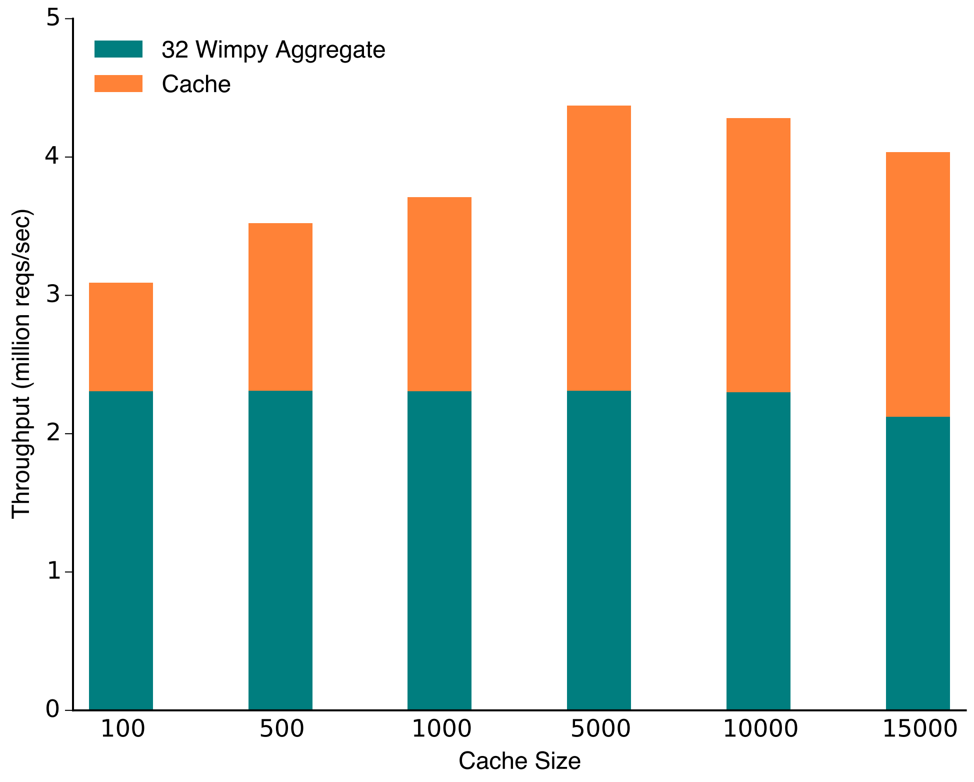


Figure 4.3: Impact of cache size on cluster throughput
If traffic skew is known, cache size can be controlled to ensure 100% utilization of each cluster node. For this setup, cache sizes > 5000 under-utilize wimpy nodes, and cache sizes < 5000 under-utilize the cache node.

4.3 Dynamic Distributions

In this section, we examine how well Exoskeleton handles changing traffic distributions. Rapidly changing traffic distributions and data spikes are an important characteristic of web-scale traffic.

We conduct an experiment to measure how fast our Exoskeleton cluster is able to react to an extreme shift in distribution. For this experiment, we measure how fast the cluster is able to return to its full capacity when a set of the most popular keys are instantaneously replaced by some of the least popular keys at regular intervals. This is accomplished simply by changing the mapping of key indexes. Figure 4.4 shows that Exoskeleton is able to rapidly recover from extreme changes in distribution. As we expect most real key popularity changes to be more gradual, we contend that Exoskeleton will be able to deal with real workload changes with far smaller impact on throughput.

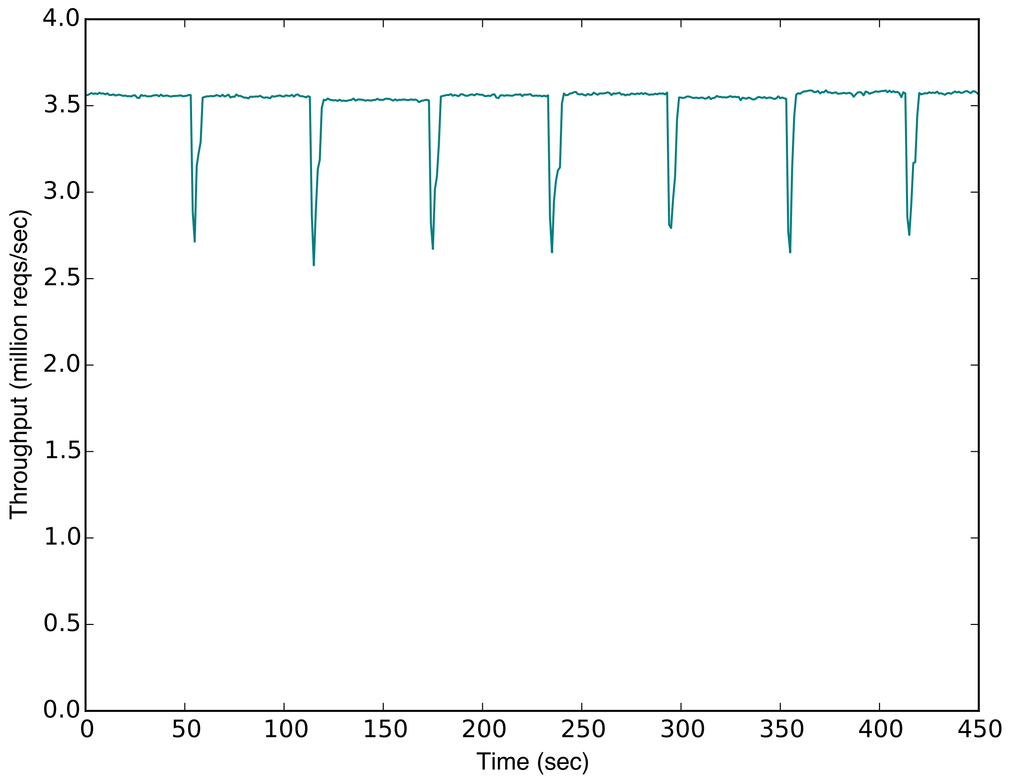


Figure 4.4: Impact of sudden workload changes on cluster throughput
Cluster throughput recovers in an average of 2.5 seconds from instantaneously replacing the 50 most popular keys with the 50 least popular keys every 50 seconds.

Chapter 5

Conclusion

5.1 Future Work

Although our results are promising, Exoskeleton is very much a work in progress. Major functionality, including replication for fault-tolerance, consistency models, alternative cache update strategies etc. need to be implemented and tested. Several optimizations and improvements must also be implemented and tested. For example, our initial experiments have not adequately captured write performance, and we have not yet tested alternative caching or consistency strategies. An intuitive next step is to replace our homegrown software with the best existing single-node key-value storage software from industry and academia. Running our cache architecture with each node running best-in-class software will allow us to thoroughly evaluate the performance on a large real-world cluster.

5.2 Conclusion

This paper demonstrates that it is possible and useful to have a cache-based key-value store cluster architecture designed from the ground up to deal with traffic from web-scale applications. We describe Exoskeleton, a high-performance prototype implementation of this architecture. Additionally, we demonstrate that Exoskeleton is able to effectively deal with the traffic characteristics of modern web-scale applications, i.e. highly-skewed distributions that change rapidly and unpredictably. Exoskeleton makes it feasible to cheaply store trillions of items on flash, and to serve this data with predictable latency and throughput regardless of traffic distributions.

Bibliography

[1] DPDK: Data Plane Development Kit. `http://dpdk.org/`.

[2] Anand Lal Shimpi. Intel SSD DC P3700 Review: The PCIe SSD Transition Begins with NVMe. `http://www.anandtech.com/show/8104/intel-ssd-dc-p3700-review-the-pcie-ssd-transition-begins-with-nvme`, 2014.

[3] Ismail Ari, Bo Hong, Ethan L. Miller, Scott A. Brandt, and Darrell DE Long. Managing flash crowds on the internet. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*, pages 246–249. IEEE.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM.

[5] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Database Systems Research (CIDR)*, volume 11, pages 223–234, 2011.

[6] Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 241–252. ACM.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154. ACM.

[8] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM.

[10] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small Cache, Big Effect: Provable Load Balancing For Randomly Partitioned Cluster Services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 23. ACM.

[11] Markus Klems, Adam Silberstein, Jianjun Chen, Masood Mortazavi, Sahaya Andrews Albert, P. P. S. Narayan, Adwait Tumbde, and Brian Cooper. The yahoo!: Cloud datastore load balancer. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, pages 33–40. ACM, 2012.

[12] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[13] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM.

[14] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[15] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle For Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM.

[16] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams. *ACM Transactions on Database Systems (TODS)*, 31(3):1095–1133, September 2006.

[17] Michael Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001.

[18] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[19] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance

storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[20] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[21] Venkata N Padmanabhan and Lili Qiu. The content and access dynamics of a busy web site: Findings and implications. *ACM SIGCOMM Computer Communication Review*, 30(4):111–123, 2000.

[22] Paul Saab. Scaling memcached at Facebook. `https://www.facebook.com/notes/facebook-engineering/scaling-memcached-at-facebook/39391378919`, 2008.

[23] Alec Wolman, M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. On the scale and performance of cooperative web proxy caching. In *ACM SIGOPS Operating Systems Review*, volume 33, pages 16–31. ACM.