Pegasus: Load-Aware Selective Replication with an In-Network Coherence Directory

Jialin Li[1], Jacob Nelson[2], Xin Jin[3], and Dan R. K. Ports[2]

[1]University of Washington, [2]Microsoft Research, [3]Johns Hopkins University

Abstract

High performance distributed storage systems face the challenge of load imbalance caused by skewed and dynamic workloads. This paper introduces Pegasus, a new storage architecture that leverages new-generation programmable switch ASICs to balance load across storage servers. Pegasus uses *selective replication* of the most popular objects in the data store to distribute load. Using a novel in-network coherence directory, the Pegasus switch tracks and manages the location of replicated objects. This allows it to achieve load-aware forwarding and dynamic rebalancing for replicated keys, while still guaranteeing data coherence. The Pegasus design is practical to implement as it stores only forwarding metadata in the switch data plane. The resulting system improves the 99% tail latency of a distributed in-memory key-value store by more than 95%, and yields up to a 9× throughput improvement under a latency SLO – results which hold across a large set of workloads with varying degrees of skewness, read/write ratio, and dynamism.

1 Introduction

Distributed storage systems are tasked with providing fast, predictable performance in spite of immense and unpredictable load. Systems like Facebook's memcached deployment [39] store trillions of objects and are accessed thousands of times on each user interaction. To achieve scale, these systems are distributed over many nodes; to achieve performance predictability, they store data primarily or entirely in memory.

A key challenge for these systems is load balancing in the presence of highly skewed workloads. Just as a celebrity may have millions of times more followers than the average user, so too do some stored objects receive millions of requests per day while others see almost none [3]. Moreover, the set of popular objects changes rapidly as new trends rise and fall. While classic algorithms like consistent hashing [23] are effective at distributing load when all objects are of roughly equal popularity, here they fall short: requests for a single popular object commonly exceed the capacity of any individual server.

Replication makes it possible to handle objects whose request load exceeds one server's capacity. Replicating *every* object, while effective at load balancing [11, 37], introduces a high storage overhead. *Selective replication* of only a set of hot objects avoids this overhead. Leveraging prior analysis of caching [14], we show that surprisingly few objects need to be replicated in order to achieve strong load-balancing properties. However, keeping track of which objects are hot and where they are stored is not straightforward, especially when the storage system may have hundreds of thousands of clients.

We address these challenges with Pegasus, a new architecture for selective replication and load balancing in a rack-scale storage system. Pegasus uses a programmable dataplane switch to route requests to servers. Drawing inspiration from CPU cache coherency protocols [4, 15, 17, 24, 26–28, 31], the Pegasus switch acts as an *in-network coherence directory* that tracks which objects are replicated and where. Leveraging the switch's view of request traffic, it can dynamically replicate or migrate data objects as the workload demands. Pegasus uses *load-aware replication* to maximize system utilization by directing read requests to the least-loaded available replica. Unlike prior approaches, Pegasus's coherence directory also allows it to dynamically rebalance the replica set *on each write operation*, accelerating both read- and write-intensive workloads – while still maintaining consistency.

Pegasus introduces several new techniques, beyond the concept of the in-network coherence directory itself. Load-aware replication requires the switch to know server load levels. We describe and evaluate two such mechanisms: (1) *reverse in-network telemetry*, where servers report their load levels to the switch, and (2) *switch-based load prediction*. We use these, along with new approximate-set-minimum structures, to implement load-aware scheduling policies in a switch dataplane.

Pegasus is a practical approach. We show that it can be implemented using a Barefoot Tofino switch, and provides effective load balancing with minimal switch resource overhead. In particular, unlike prior systems [22], Pegasus stores no application data in the switch, only metadata. This dramatically reduces switch memory usage, permitting it to co-exist with existing switch functionality and thus reducing a major barrier to adoption.

Our evaluation with 32 servers and a Pegasus switch shows:

- Pegasus reduces the 99th-percentile latency for skewed workloads by up to 97%.
- Pegasus can increase the throughput by up to 9× – or reduce by 88% the number of servers required – of a system subject to a 99%-latency SLO.
- Pegasus can react quickly to dynamic workloads where the set of hot keys changes rapidly.
- Pegasus is able to achieve these benefits for many classes

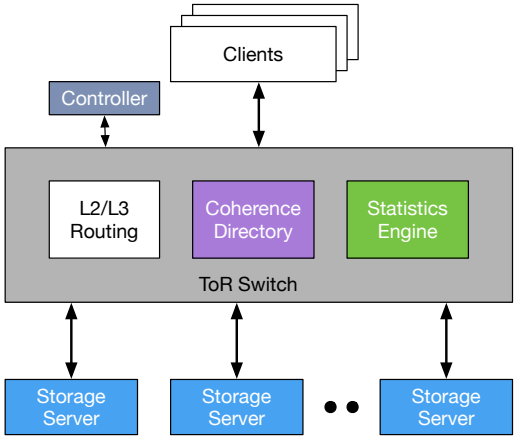


Figure 1: Pegasus architecture

of workloads, both read-heavy and write-heavy, with different levels of skew.

2 System Model

We consider a rack-scale storage system consisting of a number of storage servers connected via a single top-of-rack switch, as shown in Figure 1. The storage system stores a number of objects, each identified by a key; for simplicity, we consider a simple read/write interface. Each server is responsible for a disjoint part of the keyspace, and cross-partition operations are not possible. Pegasus specifically targets in-memory storage systems, like Redis or memcached, as these offer the fastest and most predictable performance per server.

The Pegasus architecture is a co-design of in-switch processing and an application-level protocol. This is made possible by leveraging the capabilities of newly-available switches with programmable dataplanes, such as the Barefoot Tofino, Cavium XPliant, or Broadcom Trident3 families. Broadly speaking, these chips offer reconfigurability in three relevant areas: (1) programmable parsing of application-specific headers; (2) flexible packet processing pipelines, usually consisting of 10–20 pipeline stages each capable of a match lookup and one or more ALU operations; and (3) general-purpose stateful memory, on the order of 10 MB. Importantly, all of these features are on the switch dataplane, meaning that they can be used while processing packets at full line rate – a total capacity today measured in terabits per second.

We limit ourselves here to architectures consisting of a single Pegasus switch, i.e., 32–256 servers. Larger-scale systems might be built out of multiple Pegasus deployments, with each rack responsible for a different partition of the key space.

3 Selective Replication for Load Balancing

How should a storage system handle skewed workloads, where the request load for a particularly popular object might exceed the processing capability of an individual server? Classically, two approaches have proven effective here: caching popular objects in a faster tier, and replicating objects to increase aggregate load capacity. In particular, caching has long served as the standard approach for accelerating database-backed web applications. Recent work has demonstrated, both theoretically and practically, the effectiveness of a caching approach: only a small number of keys need to be cached in order to achieve provable load balancing guarantees [14, 22, 32].

However, the effectiveness of a caching approach hinges on the ability to build a cache that can handle orders of magnitude more requests than the storage servers. Once an easily met goal, this has become a formidable challenge as storage systems themselves employ in-memory storage [39, 41, 45], new NVM technologies [19, 54], and faster network stacks [29, 33, 36]. Despite recent efforts to build faster caches out of programmable switches [22] – an approach that comes with significant practical limitations – the writing is on the wall: the era of readily-deployable magnitude-faster caches is over.

Motivated by these trends, we ask whether turning to a replication approach can provide us with a general, effective load-balancing solution. We show in this section that selective replication can provide the same provable load balancing properties as caching with neither significant space overhead nor the need to engineer a magnitude-faster cache. In §4, we show that selective replication can be implemented efficiently using an in-network coherence directory.

3.1 Caching for Load Balancing

Caching serves two performance goals: to lower latency by providing faster access to data, and to increase system throughput capacity by offloading certain requests to a cache where they can be processed faster. This paper is concerned with the second goal. We begin with a summary of recent theoretical analysis that showed that caching is particularly effective in skewed workloads [14].

Fan et al. [14] consider a storage system with n storage servers and m keys. The key space is partitioned among all storage servers, such that each of the m keys is randomly assigned to a single server. Each server has a processing capacity of r requests per second, and the total request rate the system receives never exceeds the total capacity $n \cdot r$. An adversarial workload analysis demonstrates that as long as the cache can hold the $c = O(k \cdot n \log n)$ most popular keys, the normalized load on a server is bounded by $O\left(1 + \frac{1}{\sqrt{k}}\right)$. The constant factors are not large; a cache of size $8n \log n$ ensures that each server receives no more than 20% load beyond the average. Importantly, this result depends only on the number of servers n, not the number of keys m.

A key assumption in this analysis is that the caching layer can absorb the entire load of requests to the top c most popular keys. In the extreme, this could be the entire system workload of $n \cdot r$ requests per second. As mentioned above, building a cache that can handle this workload is a daunting task.

3.2 Limitations of In-Switch Caching

Taking advantage of this load balancing result requires a cache that can handle massive system throughput, but needs only

cache a small number of objects. At first glance, caching data directly in the switch dataplane [22] appears an attractive solution, as switch ASICs are designed to sustain line rate I/O at terabits per second and are capable of processing billions of packets per second – orders of magnitude faster than server machines even with kernel-bypass I/O or RDMA. However, we observe three limitations that make in-switch caching difficult to use in practice:

First, switches have very limited on-chip memory, typically on the order of 16 to 32 megabytes [52,53]. In practice, much of this memory is used to store the L2 and L3 forwarding, ACL, VLAN, and other tables required for bread-and-butter switch functionality. When value sizes may be as large as megabytes in real world deployments [3], switch ASICs can hardly fit enough key-value pairs for effective load balancing.

Second, in-switch caches are restricted to small key-value pairs. The switch must be able to read the key and value from a packet header, but the switch's packet parser can only extract a limited-length packet header vector – a few hundred bytes at most [22]. SRAM bandwidth is also limited; each pipeline stage can only access a fixed amount. Even with a smart multi-stage design [22], the largest value size a switch can support is 128 bytes – insufficient for many workloads.

Finally, in-switch caching provides a benefit only for read-heavy workloads. Because switches can fail and do not have durable state, writes must be processed by storage servers. As a result, for workloads with a significant write fraction, the switch data plane can no longer absorb traffic for the popular keys. This issue is not just an academic one; while read-mostly workloads have attracted much attention, write-intensive and mixed workloads also commonly exist in real world deployments [3, 39].

We do not believe any of these hardware limitations are likely to change substantively in the future. On-chip SRAM is an expensive resource, and pipeline stages that read from and write to it are subject to strict timing constraints. The packet header vector size is the major factor in parser gate count, as well as in other parts of the processing pipeline [6].

3.3 Selective Replication for Load Balancing

The hardware limitations above lead us to the following requirement: *Pegasus must not store application data in the switch dataplane*. Can we nevertheless implement an effective load balancing strategy? Our key observation is that the same load balancing effect can be achieved by replicating the $O(n \log n)$ most popular keys across multiple servers rather than caching them. We show here that the same provable load balancing result applies to replication. In sections 5 to 7, we show that this selective replication approach can be implemented efficiently, storing only metadata in the switch dataplane.

Consider first a system with n nodes that handles a *read-only* workload with total request load L, and assume that each server has uniform processing capacity $r = \alpha \frac{L}{n}$, where α is a slack factor representing the maximum load imbalance. If all data were replicated on every server, i.e., any server can handle any request, then clearly there exists some way to redistribute the load such that no server exceeds its capacity (as long as $\alpha > 1$).

Can we achieve the same result even if only *some* of the keys are replicated? Fan et al's analysis says that if the most popular $O(n \log n)$ keys are cached separately from the servers, then (for the right α) no server receives load greater than its capacity. Can we re-add the load of the $O(n \log n)$ cached keys? By definition, there is enough spare capacity somewhere in the system, as the total capacity is $\alpha L > L$.

If we then replicate each of the most popular $O(n \log n)$ keys onto all n servers, we can achieve an acceptable load balancing. Since the system as a whole is underutilized, there exists at least one server whose load is below its processing capacity. We can use the following simple routing strategy: *a request for a replicated key is forwarded to the least-loaded server*.

But what about writes? A replicated write has a cost equal to the replication factor R – here, that is all nodes ($R = n$). A simple answer is to increase the slack factor to $\alpha + R f_W$, where f_W is the fraction of writes. This may be enough for read-intensive workloads. Pegasus additionally accommodates write-intensive workloads by tracking the write fraction for each object and reducing the replication factor when it is high. By choosing a number of replicas proportional to the expected number of reads per write, i.e., $R = \frac{1}{\beta f_W}$, the needed slack factor becomes $\alpha + \frac{1}{\beta}$. Strictly speaking, the analysis above does not necessarily apply in this case, as it is no longer possible to send any read to any server. However, Pegasus dynamically rebalances the replica set to the R least-loaded nodes on every write. We show empirically (§8.2) that this remains effective at load balancing. Intuitively, because the replica set is rebalanced on every write, and there are few reads between each write, the same form of load balancing continues to take place, merely on a coarser granularity.

4 A Case for In-Network Coherence Directories

We have shown in §3 that by selectively replicating a small number of popular keys, the storage system can guarantee balanced load. It remains a major challenge to track the set of replicated objects and provide strong data consistency, all without incurring significant overhead nor sacrificing load-balancing guarantees. In this section, we argue for an in-network coherence directory which manages the replicated data and guarantees data consistency with minimum overhead.

4.1 Coherence Directory for Replicated Data

Implementing selective replication poses the following challenges: first, the system needs to keep track of the replicated items and their locations (i.e., the replica set). Second, read

requests for a replicated object must be forwarded to a server in the current replica set. Third, after a write request is completed, all subsequent read requests must return the updated value.

The standard distributed systems approaches to this problem do not work well in this environment. One might try to have clients contact any server in the system, which then forwards the query to an appropriate replica for the data, as in distributed hash tables [12, 46, 47]. However, for in-memory storage systems, receiving and forwarding a request imposes nearly as much load as executing it entirely. Nor is it feasible for clients to directly track the location of each object (e.g., using a configuration service [7, 20]), as there may be hundreds of thousands or millions of clients throughout the datacenter, and it is a costly proposition to update each of them as new objects become popular or are rebalanced.

In Pegasus, we take a different approach. We note that these are the same set of challenges faced by CPU cache coherence and distributed shared memory systems. To address the above issues, these systems commonly run a cache coherence protocol using a coherence directory [4, 15, 17, 24, 26–28, 31]. For each data block, the coherence directory stores a directory entry, which contains the set of processors that have a shared or exclusive copy of the block. When a processor needs to read a data block, it sends a request to the coherence directory. The coherence directory forwards the request to a processor with a valid copy, and adds the original requestor to the sharers list. When a processor modifies a data block, it also sends a request to the coherence directory. The coherence directory either invalidates or updates all other copies, ensuring subsequent reads return the new value.

A coherence directory serves as an appropriate solution for selective replication. It can track the set of replicated objects and forward read requests to the right servers, and it can ensure data consistency by removing stale replicas from the replica set. However, to use a coherence directory for a distributed storage system requires the directory to handle all client requests. A coherence directory implemented on a conventional server will quickly become the performance bottleneck of the entire system, as well as incurring high latency overhead and unpredictable tail latency behavior.

4.2 Implementing Coherence Directory in the Network

Programmable dataplane switches provide an option that allows us to implement the coherence directory directly in the network. As detailed in §2, these switch ASICs provide flexible processing of custom-defined packet headers, and operate at full line rate. This enables us to implement a fully functional coherence directory for selective replication in the switch data plane: we store the replicated keys and their replica sets in the switch's stateful memory, program the switch to match and forward based on custom packet header fields (e.g. keys and operation types), and apply directory updating rules for the coherence protocol. We give a more detailed description of our switch implementation in §7.

Because switch ASICs are optimized for I/O, they provide the performance needed for an coherence directory. Current generation switches can support packet processing at more than 10 Tb/s aggregate bandwidth and several billion packets per second throughput [51]. Implementing the coherence directory in the top-of-rack switch for a rack-scale storage system will, almost by definition, not become the bottleneck, as the switch is designed to process packets at line rate for the entire rack. Additionally, a fast cut-through switch can process packets in a few hundred nanoseconds consistently [1], two to three orders-of-magnitude faster on average than a Linux based server system [30, 43]. Moreover, because client traffic traverses the ToR switch anyway, implementing the coherence directory in the ToR switch effectively adds zero latency overhead.

4.3 Coherence Protocol for an In-Network Coherence Directory

Designing a coherence protocol using an in-network coherence directory raises several new challenges. Traditional CPU cache coherence protocols can rely on an ordered and reliable interconnection network, and they commonly block processor requests during a coherence update. Switch ASICs have limited buffer space and therefore cannot hold packets indefinitely. Network links between ToR switches and servers are also unreliable: packets can be arbitrarily dropped, reordered, or duplicated. Implementing an ordered and reliable communication end-point on a switch is infeasible, as it requires complex logic for timeout, transmission retry, and large buffering space.

We design a new **version-based**, **non-blocking** coherence protocol to address these challenges. For each replicated object, the coherence directory stores a *current* version number, and a *next* version number. The switch increments the *next* version number on a write request and inserts it in the packet header. After the storage server processes the write request, it attaches the same *next* version number in the reply packet. The switch updates the coherence directory based on the reply's version number: if it is greater than the object's *current* version number, the switch updates the *current* version number and resets the replica set to include only the source server. The updated directory entry ensures subsequent read requests are forwarded to the server with the new value.

The above protocol still does not fully guarantee linearizability [18]. Before a write reply reaches the switch, which can be arbitrarily delayed or dropped, the switch may forward read requests to servers with either the new value or the old value. If the new value is returned to the client, and a subsequent read returns the old value, we have violated linearizability. To fix this issue, the server stores the *next* version number for each replicated object when processing write requests. It inserts the *next* version number in the reply packet of a read request. The switch then uses the same mechanism to

update the coherence directory for read replies, guaranteeing subsequent read requests will return the new value.

Our new protocol leverages two key insights. First, all storage system requests and replies have to traverse the ToR switch. We therefore only need to update the in-network coherence directory to guarantee data consistency. This allows us to avoid expensive invalidation traffic or any inter-server coordination overhead. Second, we use the monotonicity property of version numbers to handle network asynchrony. Implementing version numbers in the switch data plane – including storing, comparing, and inserting into packet headers – is made possible by the programmable switches' flexible packet processing.

4.4 Load-Aware Scheduling

As discussed in §3, to guarantee balanced load, requests for a replicated object need to be forwarded in a *load-aware* manner. More specifically, the switch needs to forward requests to servers that have not reached their processing capacity. This *load-aware* forwarding requires the switch to possess servers' load statistics. We have implemented three such mechanisms:

- **Reverse in-network telemetry.** Storage servers themselves track load statistics, e.g. CPU utilization, request rate, etc. They report load information in the reply packets going back to the switch.
- **Switch-based load prediction.** The switch estimates the current load on each server by tracking the number of outstanding requests it has forwarded to each server.
- **Randomized Forwarding.** As a baseline, we also implemented randomized forwarding. This uses no load information and may overload servers, but in some cases may suffice to provide statistical load balancing.

Applying these mechanisms to load-aware scheduling for read requests is straightforward. More surprisingly, they can also be used for write requests. At first glance it appears necessary to broadcast new writes to all servers in the replica set – potentially creating significant load and overloading some of the servers. In fact, the switch can choose a *new* replica set for the object on *each* write. It can forward write requests to one or more of the least-loaded servers, and the coherence directory ensures data consistency, no matter which server the switch selects. The ability to move data frequently allows a switch to use load-aware scheduling *for both read and write requests*. This is key to Pegasus's ability to improve performance for both read- and write-intensive workloads.

5 Pegasus Overview

We implement an in-network coherence directory and load-aware scheduling in a new rack-scale storage system, Pegasus. Figure 1 shows the high level architecture of a Pegasus deployment. All storage servers reside within a single rack. The top-of-rack (ToR) switch that connects all servers implements Pegasus's coherence directory for replicated objects.

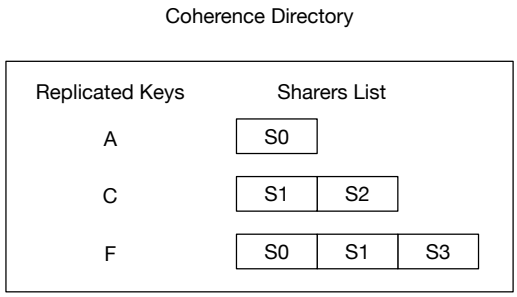


Figure 2: Logical view of the Pegasus coherence directory

Switch. The ToR switch maintains the coherence directory. Figure 2 shows a logical view of the directory: it maintains the set of replicated keys, and for each key, a list of storage servers that have a valid, shared copy of the data. To reduce switch resource overhead and to support arbitrary key sizes, the coherence directory only stores a small, fixed-size hash of each replicated key. The switch also implements a standard L2/L3 routing module and a request statistics engine.

The ToR switch identifies Pegasus packets by a reserved L3 port number. It forwards all non-Pegasus packets using standard L2/L3 routing, and thus is fully compatible with existing applications, network protocols and functions. A Pegasus packet includes a customized packet header – as shown in Figure 3 – that contains the operation type (READ, WRITE, etc.) and the hash of the requested key.

To keep space usage low, the Pegasus switch keeps directory entries only for the small set of replicated objects. Read and write requests for replicated keys are forwarded according to the Pegasus load balancing and coherence protocol. The other keys are mapped to a *home server* using a fixed algorithm, e.g., consistent hashing [23]. Although consistent hashing could be implemented in the switch, we avoid the need to do so by having clients address their packets to the appropriate server; for non-replicated keys, the Pegasus switch simply forwards them according to standard L2/L3 forwarding policies.

To handle dynamic workloads with changing key popularities, the switch also implements a request statistics engine that tracks the access rate of the replicated keys. The controller reads access statistics from the engine, and compares them with heavy hitter reports from the storage servers to find the most popular keys.

Controller. The main task of the Pegasus controller is to decide *which* keys should be replicated. It is responsible for constantly updating the coherence directory with the most popular $O(n \log n)$ keys. To do so, the controller collects heavy hitter reports from the storage servers, and compares them with access rates of replicated keys read from the switch statistics engine. The controller keeps only soft state, and therefore can be immediately replaced upon a controller failure. The controller can be deployed directly inside the ToR switch OS, or run on any remote server.

Figure 3: Pegasus packet format

Switch States:
- `rkeys`: set of replicated keys
- `rset`: map of replicated keys → set of servers with a valid copy
- `ver_curr`: map of replicated keys → next version number
- `ver_next`: map of replicated keys → current version number
- `all_servers`: set of all storage servers

Switch Functions:
- `select(p, s)`: returns one or more servers in set s based on selection policy p

Figure 4: Switch states and functions

6 Pegasus Protocol

6.1 Packet Format

Pegasus defines an application-layer packet header embedded in the L4 payload, as shown in Figure 3. Pegasus reserves a special UDP/TCP port for the switch to match Pegasus packets. The application-layer header contains an OP field, which can be one of the following operation types: READ and WRITE for client requests, or READ-REPLY and WRITE-REPLY for server replies. KEYHASH is an application-generated, fixed-size hash value of the key. VER contains an optional version number of the key for the coherence protocol.

6.2 Switch State

To implement an in-network coherence directory, Pegasus maintains a small amount of metadata in the switch dataplane, as listed in Figure 4. A lookup table `rkeys` stores the $O(n \log n)$ replicated hot keys, using KEYHASH in the packet header as the lookup key. For each replicated key, the switch maintains the set of servers which have a valid copy in `rset`. `ver_curr` and `ver_next` store each key's *current* and *next* version number respectively. The switch also maintains the set of all storage servers in `all_servers`. Additionally, the switch also implements a `select(p, s)` function that returns one or more servers in set s based on selection policy p. In §7, we elaborate how we implement the above states and functions in the switch dataplane.

6.3 Request and Reply Processing

Pegasus leverages the in-network coherence directory to distribute load for the selectively replicated objects. The switch forwards READ requests to servers in the replica set in a load-aware manner, and updates the coherence directory after a WRITE request is completed. Algorithm 1 and Algorithm 2 give the pseudo code for switch request and reply processing respectively.

6.3.1 Handling Client Requests

The switch matches the request key with entries in the `rkeys` lookup table. For replicated READs, the switch, based on the selection policy, chooses one server from the key's `rset` as the destination (Algorithm 1 line 2-3). For replicated WRITEs, the switch increments the *next* version number of the key, writes that version number into the packet header, and forwards the packet to one or more selected servers (Algorithm 1 line 4-6).

Algorithm 1 HandleRequestPacket(pkt)

```
1: if rkeys.contain(pkt.key) == true then
2:     if pkt.op == READ then
3:         pkt.dst ← select(p, rset[pkt.key])
4:     else if pkt.op == WRITE then
5:         pkt.dst ← select(p, all_servers)
6:         pkt.ver ← ++ver_next[pkt.key]
7:     end if
8: end if
9: Forward packet
```

Algorithm 2 HandleReplyPacket(pkt)

```
1: if rkeys.contain(pkt.key) == true then
2:     if pkt.ver > ver_curr[pkt.key] then
3:         ver_curr[pkt.key] ← pkt.ver
4:         rset[pkt.key] ← set(pkt.src)
5:     else if pkt.ver == ver_curr[pkt.key] then
6:         rset[pkt.key].add(pkt.src)
7:     end if
8: end if
9: Forward packet
```

Servers process READ and WRITE requests similar to existing key-value stores. To implement the version-based coherence protocol as described in §4.3, we augment the servers to store a version number for each key alongside its value. For replicated WRITEs, the switch fills in the VER header field. The server updates the key's value and version number only if the packet has a *higher* VER number; otherwise the request is dropped. The server also inserts the updated version number in the packet header of WRITE-REPLY. For replicated READs, the server reads the version number from the data store and writes it into the packet header of READ-REPLY.

6.3.2 Handling Server Replies

When the switch receives a READ-REPLY or a WRITE-REPLY, it looks up the reply's key in the switch `rkeys` table. If the key is replicated, the switch compares VER in the packet header with the *current* version number of the key. If the reply has a higher version number, the switch updates the key's *current* version number, and resets its replica set to include only the source server (Algorithm 2 line 2-4). If the two version numbers are equal, the switch adds the source server to the key's replica set (Algorithm 2 line 5-6).

The effect of this algorithm is that write requests are sent to a new replica set which may or may not overlap with the previous one. As soon as one server completes and acknowledges the write, the switch directs all future read requests to it – which is sufficient to ensure linearizability. As other replicas also acknowledge the same version of the write, they begin to

receive a share of the read request load.

6.4 Server Selection Policy

Which server should be chosen for a request? Pegasus supports two general policies: a random choice (*random*) and a least-loaded server policy (*minimum load*). For the latter, the Pegasus switch must estimate the load at each server. We have implemented and evaluated two such policies.

Reverse in-network telemetry. Our first approach relies on servers to report their load to the Pegasus switch on every reply packet – the inverse of in-network telemetry [21], which calls for *switches* to report their load to *servers*. They report this numeric value in a designated field in the Pegasus header. In a sense, this is the most general strategy, as it leaves it to the endpoints to choose what load metric to use; this flexibility makes it easier to handle heterogeneous clusters where servers may have different capacity, for example. However, as we show in §8, it suffers from a classic control loop delay problem [2]. That is, because server reply packets take time to reach the switch, the switch is making forwarding decisions based on past load information. By the time the request reaches the "least loaded" server, the server may have already received a burst of requests.

Load prediction. The second option is to predict server load on the switch, based on projected queue length at the server. The switch increments a counter each time a request is forwarded to the server. The counter could be decremented each time a reply packet is received; however, packet drops (a common occurrence on overloaded servers) make this problematic. Empirically, we have found the most effective approach to be to decrement the counter at a periodic *time interval* intended to correspond to the rate at which a server drains its message queue. To handle servers of different capacity, servers report their processing rate using the reverse in-network telemetry mechanism. This hybrid approach gives the best of both worlds; it avoids the control loop delay problem with the pure-RINT approach.

Write replication policy. Read operations are sent to exactly one replica. For write requests, the switch has the option to select any number of servers to forward it to. Larger replica set sizes improve load balancing by offering more options for future read requests. However, they also increase the cost of write operations: every server in the replica set must process the WRITE. We have seen that for workloads with a substantial write fraction, increasing the write cost for the most popular keys can easily overload the system and negate any load balancing benefit.

Our key observation is that Pegasus only needs to choose multiple recipients for WRITE operations *during read heavy workloads*, in which the load increase caused by WRITEs is negligible. For write heavy workloads, the number of consecutive READs following a WRITE is small. A single WRITE recipient is sufficient to handle to the load, until the next WRITE arrives. We exploit this observation in a simple way: the switch tracks the average number of writes per read for each key, then caps the number of replicas at this level (multiplied by a constant factor). As discussed in §3, this bounds the overhead regardless of the write fraction.

6.5 Adding and Removing Replicated Keys

Key popularities change constantly. To deal with this dynamism, the Pegasus controller continually monitors key access frequencies and updates the coherence directory with the most popular $O(n \log n)$ keys. Frequency monitoring for all keys on the switch, however, requires an infeasible amount of switch memory. To reduce memory usage, we only keep access counters for the currently replicated keys on the switch. Storage servers, with much larger memory, track access rate for the non-replicated keys, and periodically send heavy hitter reports to the controller. The controller compares heavy hitter reports with switch access counters, and updates the `rkeys` set if a non-replicated key becomes more popular.

When a new key becomes popular, the controller adds it to the coherence directory with a single entry in `rset`, the home server. This does not directly move or replicate the object; however, the next write request will move the object to a new (and larger) replica set. The controller also resets `ver_curr` and `ver_next` for the key, and adds the key to `rkeys`. It also notifies all servers that the key is now in the coherence directory.

Safely removing a replicated key is more complicated, because we must the home server has the latest version of the object before the key is removed from the coherence directory. This protocol must remain correct even if there are in-flight WRITEs and WRITE-REPLYs when the key is removed. To solve this problem, Pegasus has an interim phase where the key is migrated back to the home server. This is implemented by moving the key to a special table in the switch. While in this table, the switch processes READs and WRITEs for the key as though it were replicated replicated, except that WRITE requests are only forwarded to the home server. The controller then sends a REMOVE-REQ message to all other servers. When a server receives a REMOVE-REQ, it replies to the controller with the value and the version number of the key (if present). The server also forwards all subsequent WRITE requests for the key to the key's home server. Once the controller receives all replies, it sends the most recent version to the home server. The home server applies the value only if the version number is higher than its local store, removes the key from the set of replicated keys, and sends an acknowledgment to the controller. The controller then removes the key from the coherence directory, and broadcasts a REMOVE-COMPLETE message to all servers. When a server receives a REMOVE-COMPLETE, it can remove the key from the set of replicated keys.

A result of this protocol is that each server tracks the current set of replicated keys. If a server receives a request for a key

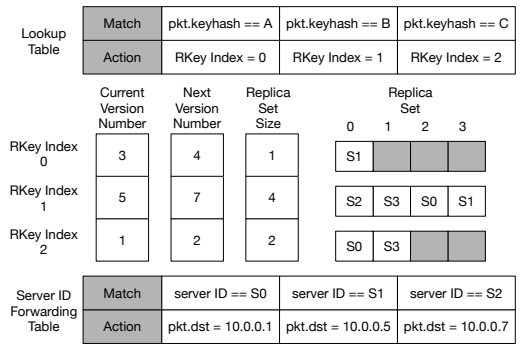


Figure 5: Coherence directory dataplane design

that is not replicated, and it is not the home server of the key, this indicates it is a delayed message (or a key hash collision; see §6.6) and should be forwarded to the key's home server.

6.6 Hash Collision

The Pegasus coherence directory acts on small key hashes, rather than full keys. Should there be a hash collision involving a replicated key, requests for non-replicated keys may be falsely forwarded to a server not its home server. Because each storage server tracks the set of currently replicated keys, it can forward the request to the correct home server. This request chaining approach has little performance impact: it only affects hash collisions *involving the small set of replicated keys*, and the requests that are forwarded are for the unreplicated, non-popular. In the extremely rare case of a hash collision involving two of the $O(n \log n)$ hot keys, Pegasus only replicates one of them to guarantee correctness.

6.7 Handling Switch Failure

Being a rack-scale storage system, failure of the Pegasus ToR switch would render the entire system unavailable. However, naively rebooting or replacing the switch could cause coherence violations, because the location of replicated keys could be lost. To address this issue, the controller polls each server for any replicated objects (and their versions) it may be storing. It then sends the value with the highest version number to each object's home server. After receiving acknowledgments from the home servers, the controller informs the switch and all servers to clear the set of replicated keys. Only then does it resume processing client requests.

7 Switch Implementation

Pegasus implements the in-network coherence directory, version-based coherence protocol, and load-aware selection policy in the dataplane of a programmable switch. This section details that implementation.

7.1 Coherence Directory

Pegasus leverages the stateful memory in the programmable switch ASICs to construct a coherence directory. Switches such as Barefoot's Tofino [50] expose their stateful memory as *register arrays*. Individual elements of an array is accessed by an *index*, and can be read and updated at line rate.

Figure 5 shows how we build a coherence directory for selective replication using exact-match tables and register arrays. First, an exact-match table is used to match the hash of the replicated keys. Each matching entry supplies an *rkey index* to be used for the register arrays. Second, a list of register arrays, one for each replicated key, store the replica set. The *rkey index* in the match table entry is used to locate the register array for the key. Third, three additional register arrays maintain the size of the replica set, the current version number, and the next version number for each replicated key. Finally, a server ID forwarding table matches on a server ID, and rewrites the packet destination address to the corresponding server on a match. The controller updates this table when server membership changes.

When the switch receives a READ request, the pipeline first matches KEYHASH in the packet header with the exact-match table. If there is a match, the switch uses the *rkey index* to locate the replica set register array. It then reads a server ID from an array element, chosen using the server selection policy (§7.2), and forwards the packet using the server ID forwarding table.

For WRITE packets that have a match in the lookup table, the switch increments the next version number in the register array (using *rkey index*), and fills the version number in the VER header field. The server selection policy for a WRITE may choose multiple servers.

For READ-REPLYs and WRITE-REPLYs that have a match in the lookup table, the pipeline first checks the VER field in the packet header with the current version number register. If VER is greater than the current version number, the switch writes the ID of the source server into the replica set register array at index zero, updates the current version number register, and changes the replica set size register to one. If VER equals the current version number, the switch increments the replica set size register, and uses the new size as an index to write the server ID into the replica set register array.

When adding or removing replicated keys, the switch controller inserts or deletes entries in the lookup table through the switch control plane interface. When a new key is added to the table, the controller also writes the ID of the home server into the corresponding replica set register array at index zero, sets the replica set size register to one, and resets the current and next version number registers.

7.2 Server Selection Policy

In §6.4, we discussed several server selection policies for load-aware scheduling. We now describe how we realize these policies in the switch dataplane.

Randomized Forwarding. Lacking a random number generator, we use a round-robin mechanism to approximate random server selection. As shown in Figure 6, we allocate a round-robin counter register. For READs, the switch increments the round-robin counter register corresponding to the

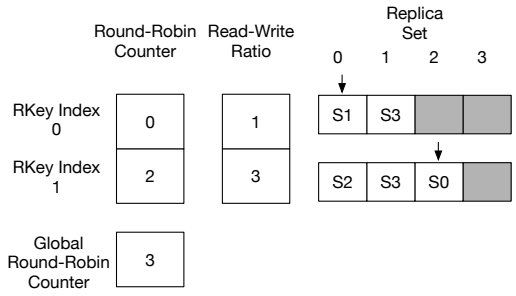


Figure 6: Random policy dataplane design

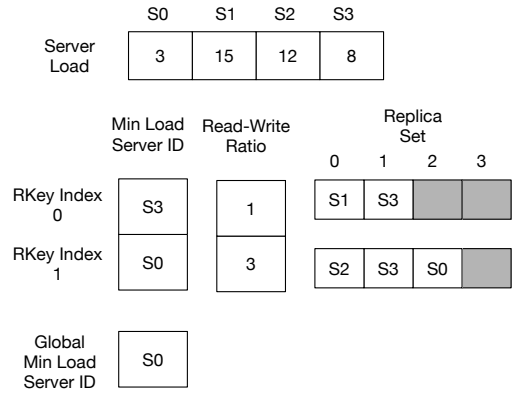


Figure 7: Min load policy dataplane design

key, and uses it to index into the replica set register array. For WRITEs, there is a read-write ratio register maintained by the statistics engine (§7.3). The pipeline reads this value to determine the replication factor. In order to send to R replicas, we statically configure a set of broadcast groups for each size R. The switch increments and uses a global round-robin counter to select a broadcast group from the set.

Minimum Load Policy. In order to select servers with the minimum load, we allocate a register array that stores the load for each server, as shown in Figure 7. If a reverse in-network telemetry policy is used, servers insert load value into the header of the reply packets. When processing reply packets, the switch updates the server load register array with the load value in the header. If instead a load prediction policy is used, the switch increments the register corresponding to the destination server each time it forwards a request packet. To decrement the register at a particular rate over time, we use the packet generator on Tofino to generate virtual DEC packets for each server. The generator determines the generation rate using a per-server processing rate updated using the reverse in-network telemetry mechanism. Processing a DEC causes the switch to decrement the corresponding server load register, but produces no output packet.

The switch data plane does not directly support finding the minimum value in a set. To naively do a pairwise comparison across elements in the register array would require require pipeline stages proportional to the number of storage servers, not a scalable approach. Instead, we approximate the minimum function by allocating a register array that tracks the least loaded server for each replica set, as well as the global minimum. This global minimum is easy to update: every time a server's load changes, the switch compares the new load against the global minimum, and updates it as needed. The per-replica-set minimum, however, requires a more elaborate update scheme: on each reply packet, the switch picks a replica set, reads the load of one of the servers in the set, and compare it to the set's minimum load register. Essentially, we use the reply packets as "probes" to update the per-replica-set minimum.

The per-replica-set minimum load register is used to choose the least loaded server for a READ. For WRITEs, we use the global minimum register to find the least loaded server. If the WRITE is forwarded to only one server, this is sufficient. Otherwise, because tracking the least loaded n servers in general is more challenging and requires more switch resources, we instead use the single least loaded server plus $n-1$ randomly selected servers.

7.3 Request Statistics Engine

The request statistics engine tracks three statistics: the access rate and read-write ratio for each replicated key, and the load for each server. The first two are computed using per-replicated-key read and write counters that are updated each time a request hits on the corresponding key. The Pegasus controller periodically reads these registers and computes the read-write ratio. To support the load prediction policy, the engine tracks each server's processing rate, and updates it based on telemetry information that the servers store in packet headers.

8 Evaluation

We implemented a prototype of Pegasus, including the switch data and control planes, a Pegasus controller, and a simple in-memory key-value store that runs the Pegasus protocol. The switch data plane is implemented in P4 [5], and compiled using the Barefoot Capilano SDE [8]. The data plane implementation runs on a Barefoot Tofino programmable switch ASIC [50]. The Pegasus controller is written in Python. It reads and updates the switch data plane through Thrift APIs [49] generated by the P4 SDE. The in-memory key-value store client and server are implemented in C++.

We ran all our experiments on a testbed that consists of ten servers with 2.5 GHz Intel Xeon E5-2680 v3 processors and 64 GB RAM running Ubuntu Linux 18.04, connected through a Tofino-based Edgecore Wedge 100BF-32 top-of-rack switch. We ran up to thirty-two key-value store servers co-located on disjoint cores of four physical servers, and used the remaining six servers to generate client load.

To evaluate the effectiveness of Pegasus under realistic workloads, we generated load using concurrent open-loop clients, with inter-arrival time following a Poisson distribu-

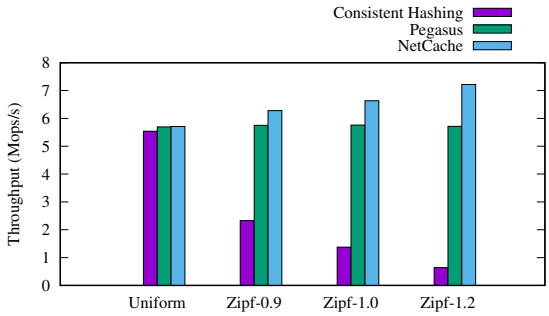


Figure 8: Throughput with a 99% latency SLO of 300 us

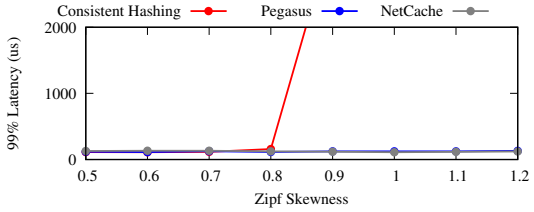
tion. The total key space consists of one million randomly generated keys, and client requests chose keys following either a uniform distribution or a Zipf distribution with different skewness parameters. For all experiments, we used 64-byte keys and 128-byte values. We also experimented with varying read/write request ratio, from read-intensive to write-intensive workloads.

We compared Pegasus against two other load balancing solutions: a conventional static consistent hashing scheme for partitioning the key space, and NetCache [22]. The NetCache implementation caches up to 10,000 128-byte values in the switch data plane, consuming more than 1 MB of switch memory. In contrary, Pegasus stores less than 5 KB of forwarding metadata, a 200× space reduction compared to NetCache.

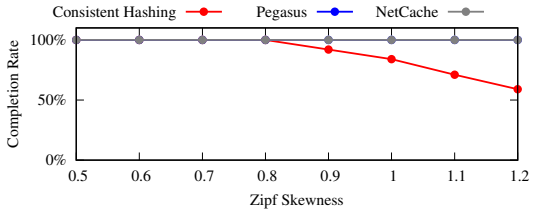
8.1 Impact of Skewness

To test and compare the performance of Pegasus under a skewed workload , we measured the maximum throughput of all three systems subject to a 99%-latency SLO. To do so, we gradually increased the client load until the 99% end-to-end latency exceeds 300μs. Figure 8 shows system throughput under increasing workload skewness with read-only requests. Pegasus maintains the same throughput level even as the workload varies from uniform to high to extreme skewness (Zipf $\alpha = 0.9$–1.2), demonstrating its effectiveness in balancing load under highly skewed access patterns. In contrast, throughput of the consistent hashing system drops to as low as 12% under more skewed workloads. At $\alpha = 1.2$, Pegasus achieves a 9× throughput improvement over consistent hashing. NetCache provides similar load balancing benefits. In fact, its throughput *increases* with skew, outperforming Pegasus. This is because requests for the cached keys are processed directly by the switch, not the storage servers, albeit at the cost of significantly higher switch resource overhead.

Figure 9 compares the 99% latency and request completion rate of the three systems with increasing Zipf skewness. We set the request rate to achieve 80% utilization on a uniform workload, and maintain this request rate as we increase skew. Both Pegasus and NetCache maintain the same low tail latency as compared to running a uniform workload, and complete 100% of the requests. Tail latency of the consistent hashing system, however, spikes when workload skewness



(a) 99% latency with increasing workload skewness



(b) Completion rate with increasing workload skewness

Figure 9: 99% latency and completion rate with increasing workload skewness

exceeds 0.9. At Zipf-1.2, Pegasus improves the tail latency over consistent hashing by 97%. There, one or more storage servers are constantly saturated, and client requests are either queued or dropped or dropped once the server receive buffers are filled. In fact, the consistent hashing system is able to complete only 60% of the client requests under Zipf-1.2.

8.2 Read/Write Ratio

Pegasus targets not only read-intensive workloads, but also write-intensive and read-write mixed workloads, both of which are common in real deployments [3]. Figure 10 shows the maximum throughput subject to a 99%-latency SLO of 300 μs, with varying read ratio. The Pegasus coherence protocol allows write requests to be processed by any storage server, so Pegasus can load balance both read and write requests. As a result, Pegasus is able to handle skewed workloads at the same throughput level as uniform ones, regardless of the read/write ratio. This is in contrast to NetCache, which can only balance read-intensive workloads; it requires storage servers to handle writes. As a result, NetCache's throughput drops rapidly as the write ratio increases, approaching the same level as static consistent hashing. Even when only 20% of requests are writes, its throughput drops by 60%. Its ability to balance load is eliminated entirely for write-only workloads. In contrast, Pegasus maintains its high throughput even for write-intensive workloads, achieving as much as 7.9× the throughput than NetCache.

8.3 Scalability

To evaluate the scalability of Pegasus, we measured the maximum throughput subject to a 99%-latency SLO under a skewed workload (Zipf 1.2) with increasing number of storage servers, and compared it against the consistent hashing system. As shown in Figure 11, Pegasus scales nearly perfectly as the number of servers increases. On the other hand, throughput of consistent hashing stops to scale after four servers: due to se-

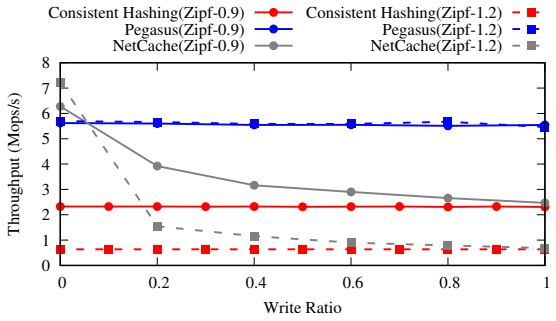


Figure 10: Throughput vs. write ratio

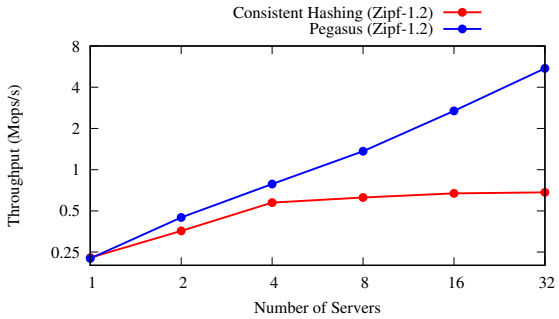


Figure 11: Scalability

vere load imbalance, the overloaded server quickly becomes the bottleneck of the entire system. Adding more servers thus does not further increase the overall throughput.

8.4 Impact of Number of Replicated Keys

The theoretical analysis in §3 proves that Pegasus needs to replicate the $O(n \log n)$ most popular keys to balance load under arbitrary access patterns. What constant factors lie hidden here? For adversarial workloads, they are not high (e.g, $8n \log n$) [14]. We show in Figure 12 that they are even lower for our non-adversarial Zipf workload. Specifically, Pegasus only needs to replicate 4, 8, and 16 keys to reach the same throughput level as running a uniform workload under Zipf distributions $\alpha = \{0.9, 1.0, 1.2\}$ respectively – significantly *less* than $n \log n$. While these numbers would be expected to increase with more servers, they easily remain within the capacity of the switch's register memory.

8.5 Load-Aware Scheduling Policies

We have implemented two policies for load-aware scheduling: *minimum load* and *random*. For the *minimum load* policy, Pegasus additionally supports two mechanisms to track server load levels: server-based load report and switch-based load prediction. We evaluated all three variations of the load-aware scheduling schemes, and Figure 13 shows the maximum throughput under increasing workload skewness respectively.

The *minimum load* policy is more effective with switch-based load prediction rather than server-based load reporting. As mentioned in §6.4, server-based load report suffers from

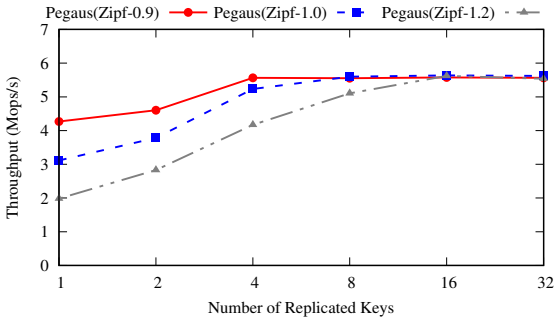


Figure 12: Throughput vs. number of replicated keys

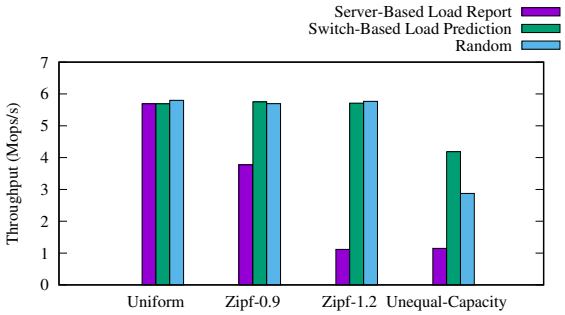


Figure 13: Comparing Pegasus server selection policies: throughput with a 99% latency SLO of 300 us

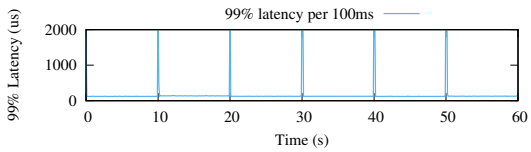
a longer control loop delay that results in temporary server overloading before the switch can react. The switch-based load prediction mechanism, on the other hand, can accurately predict server load at forwarding time, avoiding any control loop delay.

A *random* policy is in fact quite effective at distributing load when we use a set of dedicated, homogeneous servers with the same load capacity. It begins to fall short, however, when some servers are more capable than others, or background process sap their available capacity. We evaluated this by reducing the processing capacity of half of the servers by 50%. As shown in Figure 13, throughput with *random* policy drops 50% as the slower servers become the performance bottleneck, even though the faster servers still have spare processing capacity. By having the servers report their process rate, the *minimum load* policy with load prediction distributes requests in a load-aware manner, allowing both the slower and faster servers to fully utilize their processing capacity.

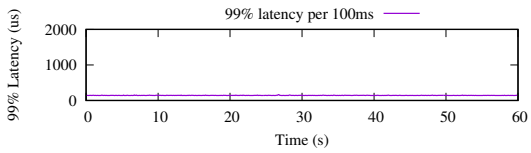
8.6 Handling Dynamic Workloads

Finally, we evaluated Pegasus under dynamic workloads with changing key popularities, similar to SwitchKV [32] and NetCache [22]. Specifically, we selected 100 keys every 10 seconds and changed their popularity rankings in the Zipf distribution. Here we consider two dynamic patterns:

- **Hot-in.** The 100 coldest keys in the popularity ranking are promoted to the top of the list, immediately turning them into the hottest objects. This workload represents extreme fluctuations in object popularities, which we hypothesize



(a) Hot-in



(b) Random

Figure 14: Dynamic workloads

is rare in real world workloads.

- **Random.** We randomly select 100 keys from the 10,000 hottest keys, and swap their popularities with another set of randomly chosen keys. As the most popular keys are less likely to be changed, this dynamism represents a more moderate change to object popularities.

We evaluate Pegasus for these workloads with a Zipf-1.2 workload and 80% utilization. We report the 99% end-to-end latency measured in time intervals of 100ms.

Hot-in. As shown in Figure 14a, sudden changes to the popularities of *all* hottest keys cause the tail latency to increase. Pegasus, however, is able to immediately detect the popularity changes and updates the in-switch coherence directory. Within 100 ms, tail latency observed by clients recovers as Pegasus resumes balancing load effectively for the new popular keys. A workload change this drastic is unlikely in practice, but Pegasus nevertheless reacts quickly.

Random. Under a *random* dynamic pattern, only a moderate number of the most popular keys are changed. Pegasus thus can continue balancing load for the unaffected replicated keys, and leveraging load-aware scheduling to avoid overloading the servers. As shown in Figure 14b, the 99% tail latency remains largely unaffected.

9 Related Work

Load Balancing. Load imbalance in large-scale key-value stores has been addressed by past systems in three ways. Consistent hashing [23] and virtual nodes [10] are widely used, but do not perform well with changing workloads. Solutions based on migration [9, 25, 48] and randomness [38] can be used to balance dynamic workloads, but these techniques introduce additional overheads and have limited ability to handle high skew. EC-Cache [44] balances load using erasure coding to split and replicate values, but works best for large keys in data-intensive clusters. SwitchKV [32] balances load across a flash-based storage layer using switches to route to an in-memory caching layer; it cannot react fast enough to changing load when the storage layer is in memory. NetCache [22] caches values directly in programmable dataplane switches; while this provides excellent throughput and latency, value sizes are limited by switch hardware constraints.

Another class of load balancers are designed to balance layer 4 traffic, such as HTTP, across a dynamic set of backend servers. These systems may be implemented as clusters of servers, as in Ananta [42], Beamer [40], and Maglev [13]; or using switches, as in SilkRoad [35] or Duet [16]. These load balancers are designed to balance long-lived flows across servers, whereas Pegasus balances load of individual request packets.

Directory-Based Coherence. Directory-based coherence protocols have been used in a variety of shared-memory multiprocessors and distributed shared memory systems [4, 15, 17, 24, 26–28, 31]. These systems can be thought of as key-value stores with fixed-size keys (addresses) and values (cache lines or pages). Directory protocols have been used in general key-value stores as well; IncBricks [34] implements an in-network key-value store using a distributed directory to cache values in network processors attached to datacenter switches. Keys have a designated home node that must be involved in writes and coherence operations, limiting the opportunity for load-balancing. In Pegasus, switches are responsible only for routing data to servers, where keys and values are stored; keys in Pegasus may be stored in any server, leading to better load-balancing.

10 Conclusion

With Pegasus, we have demonstrated that programmable switches can improve the load balancing of a storage application. Using our in-network coherence directory protocol, the switch takes over responsibility for placement of the most popular keys. This makes possible new data placement policies that cannot be achieved using traditional methods, such as reassigning the set of replicas on each write or selecting read replicas based on fine-grained load measurements. The end result is that Pegasus increases by $9\times$ the throughput level achievable subject to a latency SLO, compared to a consistent hashing workload. This permits a major reduction in the size of a cluster needed to support a particular workload.

More broadly, we believe that Pegasus provides an example of the class of applications that programmable dataplane switches are well suited for. It takes a classic use case for network devices – load balancing – and extends it to the next level by integrating it with an application-level protocol.

References

[1] Arista 7150 series.

[2] K. J. Astrom and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Princeton, NJ, USA, 2008.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, England, UK, 2012.

[4] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '90, pages 168–176, 1990.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, July 2014.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of ACM SIGCOMM 2013*, Hong Kong, China, Aug. 2013. ACM.

[7] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, USA, Nov. 2006. USENIX.

[8] Barefoot Capilano SDE. `https://www.barefootnetworks.com/products/brief-capilano/`.

[9] Y. Cheng, A. Gupta, and A. R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 4:1–4:16, New York, NY, USA, 2015. ACM.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 202–215, New York, NY, USA, 2001. ACM.

[11] J. Dean and L. A. Barosso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, USA, Oct. 2007. ACM.

[13] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 523–535, Berkeley, CA, USA, 2016. USENIX Association.

[14] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*, Cascais, Portugal, 2011.

[15] S. Frank, H. Burkhardt, and J. Rothnie. The ksr 1: bridging the gap between shared memory and mpps. In *Digest of Papers. Compcon Spring*, pages 285–294, Feb 1993.

[16] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 27–38, New York, NY, USA, 2014. ACM.

[17] D. B. Gustavson. The scalable coherent interface and related standards projects. *IEEE Micro*, 12(1):10–22, Jan. 1992.

[18] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, July 1990.

[19] M. Honda, G. Lettieri, L. Eggert, and D. Santry. PASTE: A network programming interface for non-volatile main memory. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, USA, Apr. 2018. USENIX.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, USA, June 2010. USENIX.

[21] In-band network telemetry specification. `https://p4.org/assets/INT-current-spec.pdf`.

[22] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, 2017.

[23] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of*

the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC '97), El Paso, Texas, USA, 1997.

[24] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[25] M. Klems, A. Silberstein, J. Chen, M. Mortazavi, S. A. Albert, P. Narayan, A. Tumbde, and B. Cooper. The yahoo!: Cloud datastore load balancer. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, pages 33–40, New York, NY, USA, 2012. ACM.

[26] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 302–313, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[27] J. Laudon and D. Lenoski. The sgi origin: A ccnuma highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 241–251, New York, NY, USA, 1997. ACM.

[28] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 148–159, New York, NY, USA, 1990. ACM.

[29] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, 2017.

[30] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the 5th Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, USA, Nov. 2014. ACM.

[31] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[32] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with switchkv. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI '16)*, Santa Clara, CA, 2016.

[33] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, USA, Apr. 2014. USENIX.

[34] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 795–809, New York, NY, USA, 2017. ACM.

[35] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28, New York, NY, USA, 2017. ACM.

[36] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose, CA, USA, June 2013. USENIX.

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

[38] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, Oct. 2001.

[39] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, 2013.

[40] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, Renton, WA, 2018. USENIX Association.

[41] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, Dec. 2009.

[42] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM.

[43] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 33(4), Nov. 2015.

[44] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 401–417, Berkeley, CA, USA, 2016. USENIX Association.

[45] Redis in-memory data structure store. `https://redis.io/`.

[46] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, Nov. 2001. IFIP/ACM.

[47] 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. *IEEE/ACM Transactions on Networking*, 11(1):149–160, Feb. 2003.

[48] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, Nov. 2014.

[49] Apache Thrift software framework. `https://thrift.apache.org/`.

[50] Barefoot Tofino programmable switch ASICs. `https://www.barefootnetworks.com/products/brief-tofino/`.

[51] Broadcom's Tomahawk 3 ethernet switch chip.

[52] Wedge 100bf-65x (barefoot) switch datasheet. `https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=334`.

[53] Edgecore as7512-32x (cavium xpliant) switch datasheet. `https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=68&id=129`.

[54] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, Mar. 2015. ACM.