Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories

Jialin Li, *National University of Singapore;* Jacob Nelson, *Microsoft Research;*
Ellis Michael, *University of Washington;* Xin Jin, *Johns Hopkins University;*
Dan R. K. Ports, *Microsoft Research*

https://www.usenix.org/conference/osdi20/presentation/li-jialin

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories

Jialin Li[1], Jacob Nelson[2], Ellis Michael[3], Xin Jin[4], and Dan R. K. Ports[2]

[1]National University of Singapore, [2]Microsoft Research, [3]University of Washington,
[4]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 system 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 and consistency. The Pegasus design is practical to implement as it stores only forwarding metadata in the switch data plane. The resulting system improves the throughput of a distributed in-memory key-value store by more than 10× under a latency SLO – results which hold across a large set of workloads with varying degrees of skew, read/write ratio, object sizes, 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 [50] 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 balancing load in the presence of highly skewed workloads. Just as a celebrity may have many millions more followers than the average user, so too do some stored objects receive millions of requests per day while others see almost none [3,67]. Moreover, the set of popular objects changes rapidly as new trends rise and fall [5]. While classic algorithms like consistent hashing [30] 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 [13,49], introduces a high storage overhead. *Selective replication* of only a set of hot objects avoids this overhead. Leveraging prior analysis of caching [17], 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, and keeping multiple copies consistent is even harder [50].

We address these challenges with Pegasus, a distributed storage system that uses a new architecture for selective replication and load balancing. Pegasus uses a programmable data-plane switch to route requests to servers. Drawing inspiration from CPU cache coherency protocols [4, 19, 22, 31, 34, 36, 37, 40], the Pegasus switch acts as an *in-network coherence directory* that tracks which objects are replicated and where. Leveraging the switch's central view of request traffic, it can forward requests to replicas in a load-aware manner. 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 strong consistency.

Pegasus introduces several new techniques, beyond the concept of the in-network coherence directory itself. It uses a lightweight version-based coherence protocol to ensure consistency. Load-aware scheduling is implemented using a combination of reverse in-network telemetry and in-switch weighted round-robin policy. Finally, to provide fault tolerance, Pegasus uses a simple chain replication [66] protocol to create multiple copies of data in different racks, each load-balanced with its own switch.

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 [29,45], Pegasus stores no application data in the switch, only metadata. This reduces switch memory usage to less than 3.5% of the total switch SRAM, permitting it to co-exist with existing switch functionality and thus reducing a major barrier to adoption [56].

Using 28 servers and a Pegasus switch, we show:

- Pegasus can increase the throughput by up to 10× – or reduce by 90% 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, and can recover quickly from server or rack failures.

- Pegasus can provide strong load balancing properties by only replicating a small number of objects.
- Pegasus is able to achieve these benefits for many classes of workloads, both read-heavy and write-heavy, with different object sizes and levels of skew.

2 Motivation

Real-world workloads for storage systems commonly exhibit highly skewed object access patterns [3, 6, 26, 50, 51]. Here, a small fraction of popular objects receive disproportionately more requests than the remaining objects. Many such workloads can be modeled using Zipfian access distributions [3, 5, 6, 67]; recent work has shown that some real workloads exhibit unprecedented skew levels (e.g., Zipf distributions with $\alpha > 1$) [10, 67]. Additionally, the set of popular objects changes dynamically: in some cases, the average hot object loses its popularity within 10 minutes [5].

Storage systems typically partition objects among multiple storage servers for scalability and load distribution. The implication of high skew in workloads is that load across storage servers is also uneven: the few servers that store the most popular objects will receive disproportionately more traffic than the others. The access skew is often high enough that the load for an object can exceed the processing capacity of a single server, leading to server overload. To reduce performance penalties, the system needs to be over-provisioned, which significantly increases overall cost.

Skewed workloads are diverse. Read-heavy workloads have been the focus of many recent studies, and many systems optimize heavily for them (e.g., assuming $> 95\%$ of requests are reads) [21, 29, 41, 45]. While many workloads do fall into this category, mixed or write-heavy workloads are also common [67]. Object sizes also vary widely, even within one provider. Systems may store small values (a few bytes), larger values (kilobytes to megabytes), or a combination of the two [1,3,5,67]. An ideal solution to workload skew should be able to handle all of these cases.

2.1 Existing Approaches

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? Two existing approaches have proven effective here: caching popular objects in a faster tier, and replicating objects to increase aggregate load capacity.

Caching 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 [17, 29, 41].

There are, however, two limitations with the caching approach. First, the effectiveness of caching 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 [50, 53, 58], clever data structures [42, 46], new NVM technologies [25, 68], and faster network stacks [38, 42, 48]. Recent efforts to build faster caches out of programmable switches [29, 45] address this, but hardware constraints impose significant limitations, e.g., an inability to support values greater than 128 bytes. Secondly, caching solutions only benefit read-heavy workloads, as cached copies must be invalidated until writes are processed by the storage servers.

Selective Replication Replication is another common solution to load imbalance caused by skewed workloads. By selectively replicating popular objects [2, 9, 13, 50], requests to these objects can be sent to any of the replicas, effectively distributing load across servers.

Existing selective replication approaches, however, face two challenges. First, clients must be able to identify the replicated objects and their locations – which may change as object popularity changes. This could be done using a centralized directory service, or by replicating the directory to the clients. Both pose scalability limitations: a centralized directory service can easily become a bottleneck, and keeping a directory synchronized among potentially hundreds of thousands of clients is not easy.

Providing consistency for replicated objects is the second major challenge – a sufficiently complex one that existing systems do not attempt to address it. They either replicate only read-only objects, or require users to explicitly manage inconsistencies resulting from replication [2, 9]. The solutions required to achieve strongly consistent replication (e.g., consensus protocols [35]) are notoriously complex, and incur significant coordination overhead [39], particularly when objects are modified frequently.

2.2 Pegasus Goals

The goal of our work is to provide an effective load balancing solution for the aforementioned classes of challenging workloads. Concretely, we require our system to 1) provide good load balancing for dynamic workloads with high skew, 2) work with fast in-memory storage systems, 3) handle arbitrary object sizes, 4) guarantee linearizability [24], and 5) be equally effective for read-heavy, write-heavy, and mixed read-/write workloads. As listed in Table 1, existing systems make explicit trade-offs and none of them simultaneously satisfy all five properties. In this paper, we will introduce a new distributed storage load balancing approach that makes no compromises, using an *in-network coherence directory*.

3 System Model

Pegasus is a design for rack-scale storage systems consisting of a number of storage servers connected via a single top-of-rack (ToR) switch, as shown in Figure 1. Pegasus combines in-switch load balancing logic with a new storage system. The

	Highly Skewed Workload	Fast In-Memory Store	All Object Sizes	Strong Consistency	Any Read-Write Ratio
Consistent Hashing [30]	✗	✓	✓	✗	–
Slicer [2]	✓	✗	✓	✗	–
Orleans [9]	✓	✗	✓	✗	–
EC-Cache [57]	✓	✗	✗	✓	✓
Scale-Out ccNUMA [21]	✓	✓	✓	✓	✗
SwitchKV [41]	✓	✗	✓	✓	✗
NetCache [29]	✓	✓	✗	✓	✗
Pegasus	✓	✓	✓	✓	✓

Table 1: A comparison of existing load balancing systems vs. Pegasus. In the "Any Read-Write Ratio" column, we only consider systems that provide strong consistency.

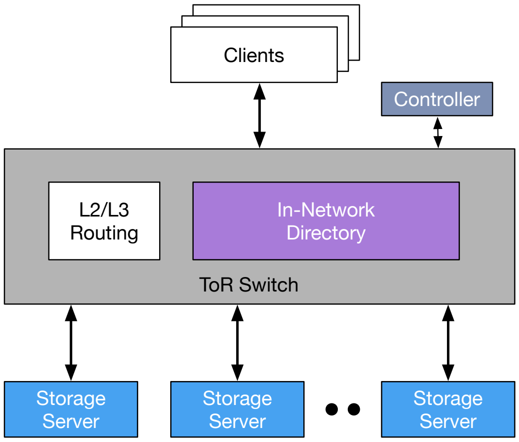


Figure 1: Pegasus system model. Pegasus is a rack-scale storage system. It augments the top-of-rack switch with an in-network coherence directory to balance load across storage servers in the rack. Servers store data in memory for fast and predictable performance.

Pegasus system provides a key-value store with a read/write interface. It does not support read-modify-write or atomic cross-key operations. Pegasus ensures strong data consistency (specifically, linearizability [24]). It uses in-memory storage to offer fast and predictable performance.

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 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.

Pegasus provides load balancing at the rack level, i.e., 32–256 servers connected by a single switch. It does not provide fault tolerance guarantees within the rack. Larger-scale, resilient systems can be built out of multiple Pegasus racks. For these systems, Pegasus ensures availability using a chain replication protocol to replicate objects across multiple racks for fault tolerance.

4 A Case for In-Network Directories

As we have discussed in §2, selectively replicating popular objects can offer good load balancing for highly skewed workloads, and it avoids certain drawbacks of a caching approach. Existing selective replication solutions, however, fall short in providing efficient directory services and strong consistency for the dynamic set of replicated objects. Our key observation is that in a rack-scale storage system (§3), the ToR switch serves as a *central point* of the system and is on the path of every client request and server reply. This enables us to implement a *coherence directory* abstraction in the ToR switch that addresses both challenges at the same time. It can track the location of every replicated object in the system and forward requests to servers with available capacity, and even change the number or location of replicas by determining where to send WRITE requests. Leveraging this in-network coherence directory, we co-design a *version-based coherence protocol* which guarantees linearizability and is highly efficient at processing object updates, enabling us to provide good load balancing even for write-intensive workloads.

4.1 Coherence Directory for Replicated Data

How do we design an efficient selective replication scheme that provides strong consistency? At a high level, the system needs to address the following challenges: first, it needs to track the replicated items and their locations with the latest value (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 [14, 59, 60]. 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 [8, 27]), 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 an object's replica set is updated.

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, 19, 22, 31, 34, 36, 37, 40]. For each data block, the coherence directory stores an entry that contains the set of processors that have a shared or exclusive copy. The directory is kept up to date as processors read and write blocks – invalidating old copies as necessary – and can always point a processor to the latest version.

A coherence directory can be applied to 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. Implemented on a conventional server, it will quickly become a source of latency and a throughput bottleneck.

4.2 Implementing Coherence Directory in the Network

Where should we implement a coherence directory that processes all client requests while not becoming a performance bottleneck? The ToR switch, as shown in Figure 1, provides a viable option for our targeted rack-scale storage systems. Switch ASICs are optimized for packet I/O: current generation switches can support packet processing at more than 10 Tb/s aggregate bandwidth and several billion packets per second [64, 65]. The programmable switches we target have a fixed-length reconfigurable pipeline, so any logic that fits within the pipeline can run at the switch's full line rate. Thus, implementing the coherence directory in the ToR switch for a rack-scale storage system will not become the bottleneck nor add significant latency, as it already processes all network traffic for the rack.

But can we implement a coherence directory efficiently in the ToR switch? To do so, two challenges have to be addressed. First, we need to implement all data structures and functional logic of a coherence directory in the switch *data plane*. We show that this is indeed possible with recent programmable switches: we store the replicated keys and their replica sets in the switch's memory, 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 detailed description of our switch implementation in §8.

Second, the switch data plane has limited resources and many are already consumed by bread-and-butter switch functionality [56]. As the coherence directory tracks the replica set for each replicated object, the switch can only support a limited number of objects to be replicated. Our design meets this challenge. Interestingly, it is possible to achieve provable load balancing guarantees if we only replicate the most popular $O(n \log n)$ objects to all servers, where n is the *number of servers* (not keys) in the system (we give a more detailed analysis of this result in §4.5). Moreover, the coherence directory only stores small metadata such as key hashes and server IDs. For a rack-scale system with 32–256 servers, the size of the coherence directory is a small fraction of the available switch resources.

4.3 A Coherence Protocol for the Network

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. Many protocols for implementing ordered and reliable communication require complex logic and large buffering space that are unavailable on a switch.

We design a new *version-based*, *non-blocking* coherence protocol to address these challenges. The switch assigns a monotonically increasing version number to each write request and inserts it in the packet header. Servers store these version numbers alongside each object, and attach the version number in each read and write reply. The switch additionally stores a *completed* version number for each replicated object in the coherence directory. When receiving read or write replies (for replicated objects), the switch compares the version in the reply with the *completed* version in the directory. If the version number in the reply is higher, the switch updates the *completed* version number and resets the replica set to include only the source server. Subsequent read requests are then forwarded to the server with the new value. When more than one server has the latest value of the object, the version number in the reply can be equal to the *completed* version. In that case, we add the source server (if not already present) to the replica set so that subsequent read requests can be distributed among up-to-date replicas.

This protocol – which we detail in §6 – guarantees linearizability [24]. It 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 inter-server coordination overhead. Second, we use version numbers, applied by the switch to packet headers, to handle network asynchrony.

4.4 Load-Aware Scheduling

When forwarding read requests, the switch can pick any of the servers currently in the replica set. The simplest policy is to select a random server from the set and rely on statistical load balancing among the servers. However, this approach falls short when the processing capacity is uneven on the storage servers (e.g. due to background tasks or different hardware specifications). To handle this issue, we also implement a weighted round-robin policy: storage servers periodically report their system load to the controller. The controller assigns weights for each server based on these load information and installs them on the switch. The switch then forwards requests to servers in the replica set proportional to their weights. Note that our in-network directory approach provides the mechanism for performing server selection. A full discussion of all scheduling policies is beyond the scope of this paper.

Surprisingly, these mechanisms 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. However, 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 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.

4.5 Feasibility of An In-Network Coherence Directory

Pegasus makes efficient use of switch resources because it only tracks object metadata (vs. full object contents [29]), and only for a small number of objects. We claimed in §4.2 that Pegasus only needs to replicate the most popular $O(n \log n)$ objects (where n is the number of servers) to achieve strong load balancing guarantees. This result is an extension of previous work [17] which showed that *caching* the $O(n \log n)$ most frequently accessed objects is sufficient to achieve provable load balancing. That is, if we exclude these objects, the remaining load on each server exceeds the average load by at most a slack factor α, which depends on the constant factors but is generally quite small; see §9.5. Intuitively, most of the load in a highly-skewed workload is (by definition) concentrated in a few keys, so eliminating that load rebalances the system.

Our approach, rather than absorbing that load with a cache, is to redistribute it among the storage servers. A consequence of the previous result is that if the total request handling capacity of the system exceeds the request load by a factor of α, then there exists a way to redistribute the requests of the top $O(n \log n)$ keys such that no server exceeds its capacity. For read-only workloads, a simple way to achieve this is to replicate these keys to all servers, then route request to any server with excess capacity, e.g., by routing a request for a replicated key to the least-loaded server in the system.

Writes complicate the situation because they must be processed by all servers storing the object. As described in §4.4, Pegasus can pick a new replica set, and a new replication factor, for an object on each write. Pegasus accomodates write-intensive workloads by tracking the write fraction for each object and setting the replication factor proportional to the expected number of reads per write, yielding constant overhead. Strictly speaking, our initial analysis (for read-only workloads) may not apply in this case, as it is no longer possible to send a read to *any* server. However, since Pegasus can rebalance the replica set on every write and dynamically adjusts the replication factor, it remains effective at load balancing for any read-write ratio. Intuitively, a read-mostly workload has many replicas, so Pegasus has a high degree of freedom for choosing a server for each read, whereas a write-mostly workload has fewer replicas but constantly rebalances them to be on the least-loaded servers.

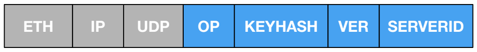


Figure 2: Pegasus packet format. The Pegasus application-layer header is embedded in the UDP payload. OP *is the request or reply type.* KEYHASH *contains the hash value of the key.* VER *is an object version number.* SERVERID *contains a unique server identifier.*

5 Pegasus Overview

We implement an in-network coherence directory 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.

Switch. The ToR switch maintains the coherence directory: it stores the set of replicated keys, and for each key, a list of servers with a valid copy of the data. To reduce switch resource overhead and to support arbitrary key sizes, the directory identifies keys by a small fixed-sized hash.

Pegasus defines an application-layer packet header embedded in the L4 payload, as shown in Figure 2. Pegasus reserves a special UDP port for the switch to match Pegasus packets. The application-layer header contains an OP field, either READ, WRITE, READ-REPLY, or WRITE-REPLY. KEYHASH is an application-generated, fixed-size hash value of the key. VER is an object version number assigned by the switch. SERVERID contains a unique identification of the server and is filled by servers on replies. If at-most-once semantics is required (§6.4), the header will additionally contain REQID, a globally unique ID for the request (assigned by the client). Non-Pegasus packets are forwarded using standard L2/L3 routing, keeping the switch fully compatible with existing network protocols.

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

Switch States:

- `ver_next`: next version number
- `rkeys`: set of replicated keys
- `rset`: map of replicated keys → set of servers with a valid copy
- `ver_completed`: map of replicated keys → version number of the latest completed WRITE

Figure 3: Switch states

other keys are mapped to a *home server* using a fixed algorithm, e.g., consistent hashing [30]. Although this algorithm 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.

Controller. The Pegasus control plane decides *which* keys should be replicated. It is responsible for updating the coherence directory with the most popular $O(n \log n)$ keys. To do so, the switch implements a request statistics engine that tracks the access rate of each key using both the data plane and the switch CPU. The controller – which can be run on the switch CPU, or a remote server – reads access statistics from the engine to find the most popular keys. The controller keeps only soft state, and can be immediately replaced if it fails.

6 Pegasus Protocol

To simplify exposition, we begin by describing the core Pegasus protocol (§6.2), under the assumption that the set of popular keys is fixed, and show that it provides linearizability. We then show how to handle changes in which keys are popular (§6.3), and how to provide exactly-once semantics (§6.4). Finally, we discuss server selection policies (§6.5) and other protocol details (§6.6).

Additionally, a TLA+ specification of the protocol which we have model checked for safety is available in our public repository [55].

6.1 Switch State

To implement an in-network coherence directory, Pegasus maintains a small amount of metadata in the switch dataplane, as listed in Figure 3. A counter `ver_next` keeps the next version number to be assigned. 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 with a valid copy in `rset`, and the version number of the latest completed WRITE in `ver_completed`. In §8, we elaborate how we store this state and implement this functionality in the switch dataplane.

6.2 Core Protocol: Request and Reply Processing

The core Pegasus protocol balances load by tracking the replica set of popular objects. It can load balance READ operations by choosing an existing replica to handle the request, and can change the replica set for an object by choosing which replicas process WRITE operations. Providing this load balancing while ensuring linearizability requires making sure that the in-network directory tracks the location of the *latest successfully written value* for each replicated key. Pegasus does this by assigning version numbers to incoming requests and monitoring outgoing replies to detect when a new version has been written.

Algorithm 1 HandleRequestPacket(pkt)

```
 1: if $pkt.op = $ WRITE then
 2:     $pkt.ver \leftarrow ver\_next$++
 3: end if
 4: if $rkeys.contains(pkt.keyhash)$ then
 5:     if $pkt.op = $ READ then
 6:         $pkt.dst \leftarrow$ select replica from $rset[pkt.keyhash]$
 7:     else if $pkt.op = $ WRITE then
 8:         $pkt.dst \leftarrow$ select from all servers
 9:     end if
10: end if
11: Forward packet
```

Algorithm 2 HandleReplyPacket(pkt)

```
1: if $rkeys.contains(pkt.keyhash)$ then
2:     if $pkt.ver > ver\_completed[pkt.keyhash]$ then
3:         $ver\_completed[pkt.keyhash] \leftarrow pkt.ver$
4:         $rset[pkt.keyhash] \leftarrow set(pkt.serverid)$
5:     else if $pkt.ver = ver\_completed[pkt.keyhash]$ then
6:         $rset[pkt.keyhash].add(pkt.serverid)$
7:     end if
8: end if
9: Forward packet
```

6.2.1 Handling Client Requests

The Pegasus switch assigns a version number to every WRITE request, by writing `ver_next` into its header and incrementing `ver_next` (Algorithm 1 line 1-3). It determines how to forward a request by matching the request's key hash with the `rkeys` table. If the key is not replicated, the switch simply forwards the request to the original destination – the home server of the key. For replicated keys, it forwards READ requests by choosing one server from the key's `rset`. For replicated WRITEs, it chooses one or more destinations from the set of all servers. In both cases, this choice is made according to the server selection policy (§6.5).

Storage servers maintain a version number for each key alongside its value. When processing a WRITE request, the server compares VER in the header with the version in the store, and updates both the value and the version number *only if* the packet has a *higher* VER. It also includes the version number read or written in the header of READ-REPLY and WRITE-REPLY messages.

6.2.2 Handling Server Replies

When the switch receives a READ-REPLY or a WRITE-REPLY, it looks up the reply's key hash in the switch `rkeys` table.

If the key is replicated, the switch compares VER in the packet header with the latest completed version of the key in `ver_completed`. If the reply has a higher version number, the switch updates `ver_completed` and resets the key's replica set to include only the source server (Algorithm 2 line 1-4). If the two version numbers are equal, the switch adds the source server to the key's replica set (Algorithm 2 line 5-7).

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.

6.2.3 Correctness

Pegasus provides linearizability [24]. The intuition behind this is that the Pegasus directory monitors all traffic, and tracks where the latest observed version of a key is located. As soon as any client sees a new version of the object – as indicated by a READ-REPLY or WRITE-REPLY containing a higher version number – the switch updates the directory to send future read requests to the server holding that version.

The critical invariant is that the Pegasus directory contains at least one address of a replica storing a copy of the latest write to be *externalized*, as well as a version number of that write. A write is externalized when its value can be observed outside the Pegasus system, which can happen in two ways. The way a write is usually externalized is when a WRITE-REPLY is sent, indicating that the write has been completed. It is also possible, if the previous and current replica set overlap, that a server will respond to a concurrent READ with the new version before the WRITE-REPLY is delivered. Pegasus detects both cases by monitoring both WRITE-REPLY and READ-REPLY messages, and updating the directory if VER exceeds the latest known compatible version number.

This invariant, combined with Pegasus's policy of forwarding reads to a server from the directory's replica set, is sufficient to ensure linearizability:

- WRITE operations can be ordered by their version numbers.
- If a READ operation r is submitted after a WRITE operation w completes, then r comes after w in the apparent order of operations because it is either forwarded to a replica with the version written by w or a replica with a higher version number.
- If a READ operation r_2 is submitted after another READ r_1 completes, then it comes after r_1 in the apparent order of operations, because it will either be forwarded to a replica with the version r_1 saw or a replica with a newer version.

6.3 Adding and Removing Replicated Keys

Key popularities change constantly. The Pegasus controller continually monitors access frequencies and updates the coherence directory with the most popular $O(n \log n)$ keys. We elaborate how access statistics are maintained in §8.

When a new key becomes popular, Pegasus must create a directory entry for it. The Pegasus controller does this by adding the key's home server to `rset`. It also adds a mapping for the key in `ver_completed`, associating it with $\texttt{ver\_next} - 1$, the largest version number that could have been assigned to a write to that key at the key's home server. Finally, the controller adds the key to `rkeys`. This process does not immediately move or replicate the object. However, later WRITE requests will be sent to a new (and potentially larger) replica set, with a version number necessarily larger than the one added to the directly. Once these newly written values are externalized, they will added to the directory as normal.

Transitioning a key from the replicated to unreplicated state is similarly straight-forward. The controller simply marks the switch's directory entry for transition. The next WRITE for that key is sent to its home server; once the matching WRITE-REPLY is received, the key is removed from the directory.

Read-only objects and virtual writes. The protocol above only moves an object to a new replica set (or back to its home node) on the next write. While this simplifies design, it poses a problem for objects that are read-only or modified infrequently. Conceptually, Pegasus addresses this by performing a write that does not change the object's value when an object needs to be moved. More precisely, the controller can force replication by issuing a *virtual write* to the key's home server, instructing it to increment its stored version number to the one in `ver_completed` and to forward that value to other replicas so that they can be added to `rset` and assist in serving reads.

6.4 Avoiding Duplicate Requests

At-most-once semantics, where duplicated or retried write requests are not reexecuted, are desirable. There is some debate about whether these semantics are required by linearizability or an orthogonal property [18, 28], and many key-value stores do not have this property. Pegasus accommodates both camps by optionally supporting at-most-once semantics.

Pegasus uses the classic mechanism of maintaining a table of the most recent request from each client [43] to detect duplicate requests. This requires that the same server process the original and the retried request, a requirement akin to "stickiness" in classic load balancers. A simple way to achieve this would be to send each write request initially to the object's home server. However, this sacrifices load balancing of writes.

We instead provide duplicate detection without sacrificing load balancing by noticing that it is not necessary for one server to see all requests for an object – only that a retried request goes to the same server that previously processed it. Thus, Pegasus forwards a request initially to a single *detector node* – a server deterministically chosen by the *request's* unique REQID, rather than the key's hash. It also writes into a packet header the other replicas, if any, that the request should be sent to. The detector node determines if the request is a duplicate; if not, it processes it and forwards the request to the other selected servers.

Some additional care is required to migrate client table state when a key transitions from being unpopular to popular and vice versa. We can achieve this by pausing WRITEs to the key during transitions. When a new key becomes popular, the controller retrieves existing client table entries from the home server and propagates them to all servers. When a popular key becomes unpopular, it queries all servers to obtain their client tables, and sends their aggregation (taking the latest entry for each client) to the home server. Once this is complete, the system can resume processing WRITEs for that key.

6.5 Server Selection Policy

Which replica should be chosen for a request? This is a policy question whose answer does not impact correctness (i.e., linearizability) but determines how effective Pegasus is at load balancing. As described in §4.4, we currently implement two such policies. The first policy is to simply pick a random replica and rely on statistical load balancing. A more sophisticated policy is to use weighted round-robin: the controller assigns weights to each server based on load statistics it collects from the servers, and instructs the switch to select replicas with frequency proportional to the weights.

Write replication policy. Read operations are sent to exactly one replica. Write requests can be sent to one or more servers, whether they are in the current replica set or not. Larger replica set sizes improve load balancing by offering more options for future read requests, but increase the cost of write operations. For write-heavy workloads, increasing the write cost can easily negate any load balancing benefit.

As discussed in §4.5, the switch tracks the average READs per WRITE for each replicated object. By choosing a replication factor to be proportional to this ratio, Pegasus can bound the overhead regardless of the write fraction.

6.6 Additional Protocol Details

Hash collisions. The Pegasus coherence directory acts on small key hashes, rather than full keys. Should there be a hash collision involving a replicated key and a non-replicated key, requests for the non-replicated key may be incorrectly forwarded to a server that is not its home server. To deal with this issue, each server tracks the set of all currently replicated keys (kept up to date by the controller per §6.3). Armed with this information, a server can forward the improperly forwarded 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*. Moreover, we only forward requests for the unreplicated keys which have low access rate. In the extremely rare case of a hash collision involving two of the $O(n \log n)$ most popular keys, Pegasus only replicates one of them to guarantee correctness.

Version number overflow. Version numbers must increase monotonically. Pegasus uses 64-bit version numbers, which makes overflow unlikely: it would require processing transactions at the full line rate of our switch for over 100 years. Extremely long-lived systems, or ones that prefer shorter version numbers, can use standard techniques for version number wraparound.

Garbage collection. When handling WRITEs for replicated keys, Pegasus does not explicitly invalidate or remove the old version. Although this does not impact correctness – the coherence directory forwards all requests to the latest version – retaining obsolete copies forever wastes storage space on servers. We handle this issue through garbage collection. The Pegasus controller already notifies servers about which keys are replicated, and periodically reports the last-completed version number. Each server, then, can detect and safely remove a key if it has an obsolete version, or if the key is no longer replicated (and the server is not the home node for that key).

7 Beyond a Single Rack

Thus far, we have discussed single-rack, single-switch Pegasus deployments. Of course, larger systems need to scale beyond a single rack. Moreover, the single-rack architecture provides no availability guarantees when servers or racks fail: while Pegasus replicates popular objects, the majority of objects still have just one copy. This choice is intentional, as entire-rack failures are common enough to make replicating objects within a rack insufficient for real fault tolerance.

We address both issues with a multi-rack deployment model where each rack of storage servers and its ToR switch runs a separate Pegasus instance. The workload is partitioned across different racks, and chain replication [66] is used to replicate objects to multiple racks. Object placement is done using two layers of consistent hashing. A global configuration service [8, 27] maps each range of the keyspace to a *chain* of Pegasus racks. Within each rack, these keys are mapped to servers as in §5. In effect, each key is mapped to a chain of servers, each server residing in a different rack.

We advocate this deployment model because it uses in-switch processing only in the ToR switches in each rack. The remainder of the datacenter network remains unmodified, and in particular it does not require any further changes to packet routing, which has been identified as a barrier to adoption for network operators [56]. A consequence is that it cannot load balance popular keys across different racks. Our simulations, however, indicate that this effect is negligible at all but the highest workload skew levels: individual servers are easily overloaded, but rack-level overload is less common.

Replication Protocol. As in the original chain replication, clients send WRITEs to the head server in the chain. Each server forwards the request to the next in the chain, until reaching the tail server, which then replies to the client. Clients send READs to the tail of the chain; that server responds directly to the client. In each case, if the object is a popular one in that rack, the Pegasus switch can redirect or replicate it.

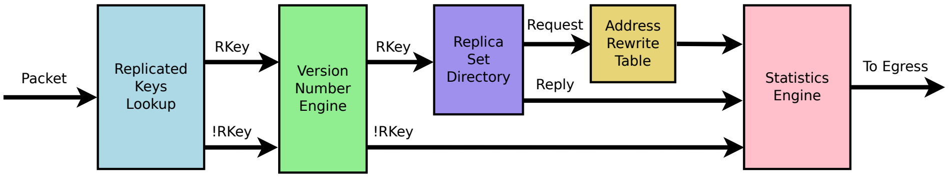


Figure 4: Switch data plane design for the Pegasus coherence directory

Pegasus differs from the original chain replication protocol in that it cannot assume reliable FIFO channels between servers. To deal with network asynchrony, it reuses the version numbers provided by the Pegasus switches to ensure consistency. Specifically, we augment Algorithm 1 in the following way: when a Pegasus switch receives a WRITE request, it only stamps `ver_next` into the request if VER in the packet header is not null; otherwise, it leaves the version number in the request unmodified and sets its `ver_next` to be one greater than that value (if it isn't already). The effect of this modification is that WRITE requests only carry version numbers from the head server's ToR switch; and the number does not change when propagating along the chain. This ensures that all replicas apply WRITEs in the same order.

Reconfiguring the Chains. If a Pegasus rack fails, it can be replaced using the standard chain replication protocol [66]. When the failure is noted, the configuration service is notified to remove the failed rack from all chains it participated in, and to add a replacement. This approach leverages the correctness of the underlying chain replication protocol, treating the Pegasus rack as functionally equivalent to a single replica.

If a system reconfiguration changes the identity of the head rack for a key range, subsequent WRITEs will get version numbers from a different head switch. If the new head rack was present in the old chain, these version numbers will necessarily be greater than any previously completed writes. If a rack is added to a chain as the head, the `ver_next` in the rack's switch must first be updated to be greater than or equal to the other switches in the chain.

If an individual server fails, a safe solution is to treat its entire rack as faulty and replace it accordingly. While correct, this approach is obviously inefficient. Pegasus has an optimized reconfiguration protocol (omitted due to space constraints) that only moves data that resided on the failed server.

8 Switch Implementation

The coherence directory (§6) plays a central role in Pegasus: it tracks popular objects and their replica sets; it distributes requests for load balancing; it implements the version-based coherence protocol; and it updates the set of replicated objects based on dynamic workload information. In this section, we detail the implementation of the Pegasus coherence directory in the data plane of a programmable switch.

8.1 Switch Dataplane Implementation

Figure 4 shows the schematic of the data plane design for the coherence directory. When a Pegasus packet enters the switch ingress pipeline, a lookup table checks if it refers to a replicated object. The packet then traverses a version number engine and a replica set directory, which implement the version-based coherence protocol (Algorithms 1 and 2). For request packets, one or more servers are selected from the replica set directory, and the packet's destination is updated by an address rewrite table. Finally, all Pegasus packets go through a statistics engine before being routed to the egress pipeline.

We leverage two types of stateful memory primitives available on programmable switching ASICs (such as Barefoot's Tofino [63]) to construct the directory: exact-match lookup tables and register arrays. A lookup table matches fields in the packet header and performs simple actions such as arithmetics, header rewrites, and metadata manipulations. Lookup tables, however, can only be updated from the control plane. Register arrays, on the other hand, are accessed by an *index* and can be read and updated at line rate in the data plane. The rest of the section details the design of each component.

Replicated Keys Lookup Table When adding replicated keys (§6.3), the controller installs its KEYHASH in an exact-match lookup table. The table only needs to maintain $O(n \log n)$ entries, where n is the number of servers in the rack. The switch matches each Pegasus header's KEYHASH with entries in the table. If there is a match, the index number associated with the entry is stored in the packet metadata to select the corresponding replicated key in later stages.

Version Number Engine We use two register arrays to build the version number engine, as shown in Figure 5. The first register array contains a single element – the next version number. If the packet is a WRITE request, the version number in the register is incremented and the switch writes the version into the packet header. The second register array stores the completed version number for each replicated key, and uses numeric ALUs to compare and update the version number (per Algorithm 2). The comparison result is passed to the next stage.

Replica Set Directory As shown in Figure 6, we build the replica set directory using three register arrays to store: (i) the size of each replica set, (ii) a bitmap indicating which servers are currently in each replica set, and (iii) a list of server IDs in each set. When choosing replicas for Pegasus READ requests,

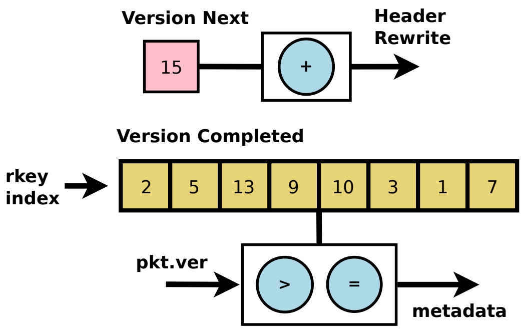


Figure 5: Design of the switch version number engine. One register array with a single element tracks the next version number. The register is incremented on each WRITE*. A second register array stores the latest completed version number for each replicated object. Numeric ALUs compare values in this array with version numbers in the reply headers.*

the replica set size is read and fed into the selection logic unit to calculate an index for locating the server ID in the list (the selection logic can pick any servers for WRITE requests). Note that we collapse the server ID list of all replicated keys into a single register array, leveraging the fact that each key can be replicated on at most n servers. Therefore, to locate the ith replica for the kth key, the index is calculated as $k*n+i$ (for brevity, we will use relative indices, i.e. i in the formula, for the remaining discussion).

If the version number engine indicates that the Pegasus reply has a higher version number, the size of the replica set is reset to one, and the replica set bitmap and server ID list are reset to contain only the server ID in the reply packet. If the version number engine indicates that the version numbers are equal, the switch uses the bitmap to check if the server is already in the replica set. If not, it updates the bitmap, increments the replica set size, and appends the server ID to the end of the server list.

To add a new replicated key, the controller sets the replica set size to one, and the bitmap and server ID list to contain only the home server.

Address Rewrite Table The address rewrite table maps server IDs to the corresponding IP addresses and ports, and is kept up to date by the controller as servers are added. When the replica set directory chooses a single server as the destination, the address rewrite table updates the headers accordingly. If the replica set directory selects multiple servers (for a WRITE request), we use the packet replication engine on the switch to forward the packet to the corresponding multicast group.

Statistics Engine To detect the most popular $O(n \log n)$ keys in the workload, we construct a statistics engine to track the access rate of each key. For replicated keys, the switch maintains counters in a register array. This approach is obviously not feasible for the vast number of non-popular keys. The statistics engine instead samples requests for unreplicated keys, forwarding them to the switch CPU. A dedicated program on the switch CPU constructs an access frequency table from sampled packets. The sampling component serves two purposes: it reduces the traffic to the switch CPU and it functions as a high-pass filter to filter out keys with low access frequency. The controller scans both statistics tables to determine when newly popular keys need to be replicated or replication stopped for existing keys, and makes these changes following the protocol in §6.3.

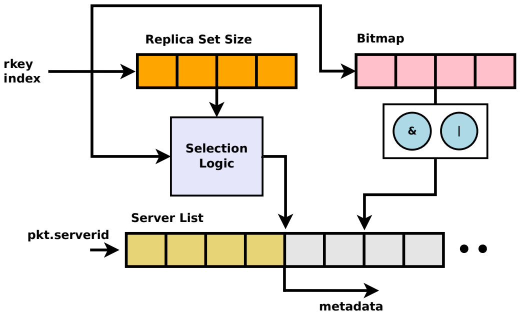


Figure 6: Design of the switch replica set directory. The directory uses three register arrays: one array stores the size of each replica set; another array maintains a bitmap for each set, tracking which servers are currently in the set; the last array stores a list of server IDs in each set.

Two separate register arrays track the READ and WRITE count for each replicated key. The controller uses these to compute the read/write ratio, which the selection logic in the replica set directory uses to decide how many replicas to use for each WRITE request.

9 Evaluation

Our Pegasus implementation includes switch data and control planes, a Pegasus controller, and an in-memory key-value store. The switch data plane is implemented in P4 [7] and runs on a Barefoot Tofino programmable switch ASIC [63]. The Pegasus controller is written in Python. It reads and updates the switch data plane through Thrift APIs [62] generated by the P4 SDE. The key-value store client and server are implemented in C++ with Intel DPDK [15] for optimized I/O performance.

Our testbed consists of 28 nodes with dual 2.2 GHz Intel Xeon Silver 4114 processors (20 total cores) and 48 GB RAM running Ubuntu Linux 18.04. These are connected to an Arista 7170-64S (Barefoot Tofino-based) programmable switch using Mellanox ConnectX-4 25 Gbit NICs. 16 nodes act as key-value servers and 12 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 distribution. The total key space consists of one million randomly generated keys, and client requests chose keys following either a uniform distribution or a skewed (Zipf) distribution.

We compared Pegasus against two other load balancing solutions: a conventional static consistent hashing scheme for partitioning the key space, and NetCache [29]. The consis-

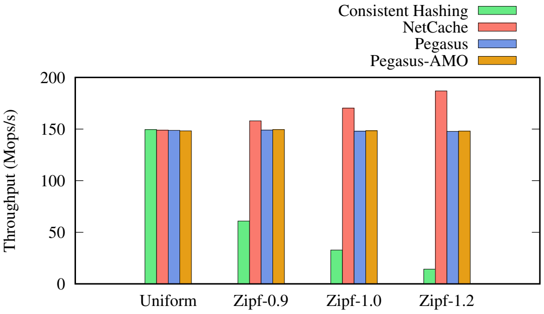


Figure 7: Maximum throughput achievable subject to a 99% latency SLO of 50 us. Pegasus successfully rebalances request load, maintaining similar performance levels for uniform and skewed workloads.

tent hashing scheme assigns 16 virtual nodes to each storage server to improve load balancing. We additionally evaluated a version of Pegasus that supports at-most-once semantics (Pegasus-AMO, as described in §6.4). To allow a comparison with NetCache, we generally limit ourselves to 64-byte keys and 128-byte values, as this is the largest object value size it can support. NetCache reserves space for up to 10,000 128-byte values in the switch data plane, consuming a significant portion of the switch memory. In contrast, Pegasus consumes less than 3.5% of the total switch SRAM. At larger key and value sizes, Pegasus maintains similar performance and memory usage, whereas NetCache cannot run at all.

9.1 Impact of Skew

To test and compare the performance of Pegasus under a skewed workload, we measured the maximum throughput of all four systems subject to a 99%-latency SLO. We somewhat arbitrarily set the SLO to 5× of the median unloaded latency (we have seen similar results with different SLOs). Figure 7 shows system throughput under increasing workload skew with read-only requests. Pegasus maintains the same throughput level even as the workload varies from uniform to high to extreme skew (Zipf $\alpha = 0.9$–1.2),[1] demonstrating its effectiveness in balancing load under highly skewed access patterns. Since the workload is read-only, Pegasus with at-most-once support (Pegasus-AMO) has the exact same performance. In contrast, throughput of the consistent hashing system drops to as low as 10% under more skewed workloads. At $\alpha = 1.2$, Pegasus achieves a 10× 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.

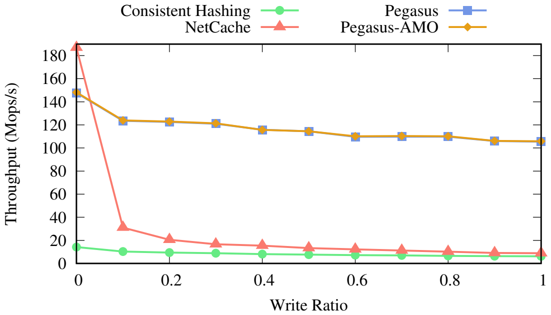


Figure 8: Throughput vs. write ratio. Pegasus maintains its load balancing advantage across the spectrum of write ratios, whereas NetCache suffers a significant penalty with even 10% writes.

9.2 Read/Write Ratio

Pegasus targets not only read-intensive workloads, but also write-intensive and read-write mixed workloads. Figure 8 shows the maximum throughput subject to a 99%-latency SLO of 50 μs when running a highly skewed workload (Zipf-1.2), with varying write ratio. The Pegasus coherence protocol allows write requests to be processed by any storage server while providing strong consistency, so Pegasus can load balance both read and write requests. As a result, Pegasus is able to maintain a high throughput level, regardless of the write ratio. Even with at-most-once semantics enforced, Pegasus-AMO performs equally well for all write ratios, by leveraging the randomness in requests' REQID (§6.4) to distribute write requests to all servers. 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 10% of requests are writes, its throughput drops by more than 80%. Its ability to balance load is eliminated entirely for write-intensive workloads. In contrast, Pegasus maintains its high throughput even for write-intensive workloads, achieving as much as 11.8× the throughput as NetCache. Note that Pegasus's throughput does drop with higher write ratio. This is due to the increasing write contention and cache invalidation on the storage servers.

9.3 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 9, Pegasus scales nearly perfectly as the number of servers increases. On the other hand, throughput of consistent hashing stops scaling after two servers: due to severe load imbalance, the overloaded server quickly becomes the bottleneck of the entire system. Adding more servers thus does not further increase the overall throughput.

We also evaluate the performance of an end-host coherence directory implementation, using Pegasus's protocol with

[1] Although $\alpha = 1.2$ is a very high skew level, some major storage systems reach or exceed this level of skew. For example, more than half of Twitter's in-memory cache workloads can be modeled as Zipf distributions with $\alpha > 1.2$ [67], as can Alibaba's key-value store workload during peak usage periods [10].

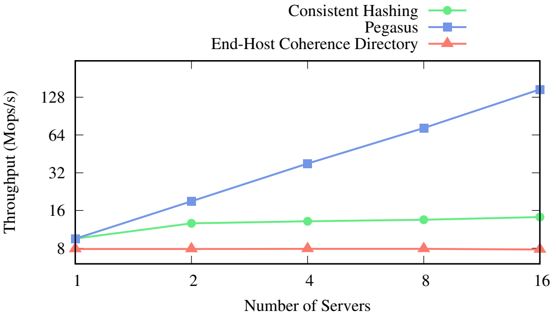


Figure 9: Scalability. Pegasus scales nearly linearly up to 16 servers, as no individual server becomes a bottleneck, even with a skewed workload.

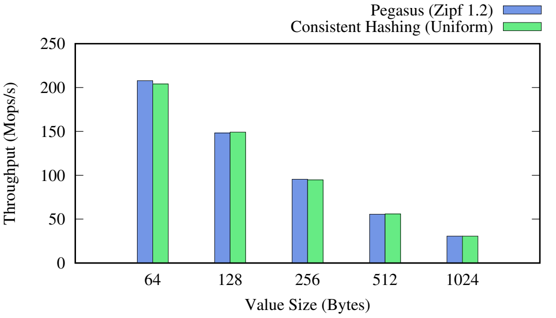


Figure 10: Throughput vs. object size. Pegasus provides effective load balancing across a wide range of object sizes. The performance of a traditional design under a uniform workload is shown as a baseline.

a server in place of the switch. Because the directory needs to process twice as many packets as the storage servers for each client request (both requests and replies), this implementation is unable to keep up with even a single server – highlighting the importance of using an accelerated platform like a switching ASIC as the coherence directory.

9.4 Object Sizes

To test if Pegasus can handle different object sizes, we varied the value size from 64 bytes to 1 KB and measured the maximum throughput of Pegasus subject to a 99%-latency SLO under the same skewed workload. We additionally plot the throughput of the consistent hashing system under a uniform workload. Figure 10 shows that Pegasus is equally efficient in load balancing for both small and large objects. Its throughput under a highly skewed workload is virtually equivalent to that of consistent hashing under a zero-skewed workload. Note that the throughput in the figure uses number of operations per second (which should naturally decrease with larger object size), not bits per second.

9.5 Impact of Number of Replicated Keys

Keeping the size of coherence directory small is crucial as switches are highly resource constrained. Our analysis (§4.5) shows that Pegasus only needs to replicate the $O(n\log n)$ most popular keys to balance load under arbitrary access patterns.

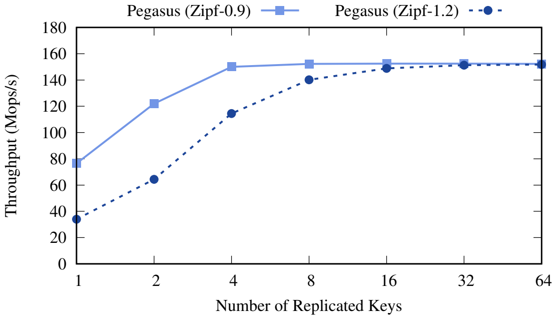


Figure 11: Throughput vs. number of replicated keys. For these workloads, only 8–16 replicated keys are needed to achieve most of Pegasus's load balancing benefit.

What constant factors are hidden here? For adversarial workloads, they are not high (e.g, $8n\log n$) [17]. We show in Figure 11 that they are even lower for our non-adversarial Zipf workload. Specifically, Pegasus only needs to replicate 8–16 keys to achieve its throughput benefit – 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.

9.6 Server Selection Policies

We have implemented two policies for selecting servers for replicated objects: random and weighted round-robin. We evaluated both policies: Figure 12 shows their maximum throughput under different workloads.

Both policies are quite effective at distributing load for uniform and highly skewed workloads when we use a set of dedicated, homogeneous servers with the same load capacity. The random policy 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 12, throughput with the random policy drops 50% as the slower servers become the performance bottleneck, even though the faster servers still have spare processing capacity. By collecting load information from the servers and setting the weights accordingly, the weighted round-robin policy allows both the slower and faster servers to fully utilize their processing capacity.

9.7 Handling Dynamic Workloads

Finally, we evaluated Pegasus under dynamic workloads with changing key popularity, similar to SwitchKV [41] and NetCache [29]. 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 is rare in real world workloads.

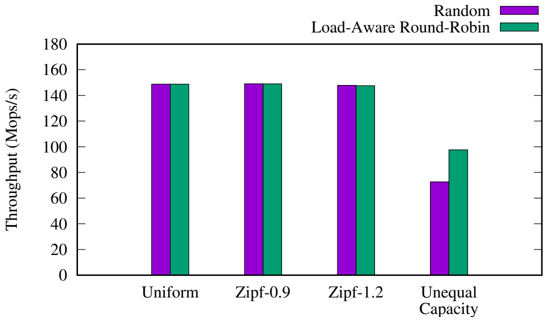
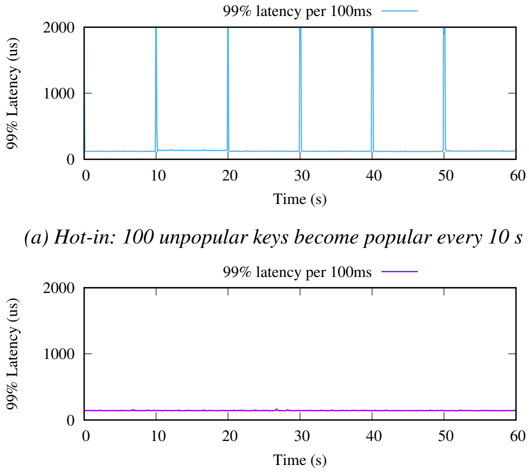


Figure 12: Comparing Pegasus server selection policies: throughput with a 99% latency SLO of 50 us. A random selection policy provides good statistical load balancing when server capacity is uniform; Pegasus's load-aware policy outperforms it otherwise.



(a) Hot-in: 100 unpopular keys become popular every 10 s

(b) Random: 100 random keys swap popularity every 10 s

Figure 13: Dynamic workloads. Pegasus reacts quickly to changes in object popularity.

- **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 popularity.

We evaluate Pegasus for these workloads with a Zipf-1.2 workload and 80% utilization.

Hot-in. Sudden changes to the popularity 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. A workload change this drastic is unlikely, but Pegasus nevertheless reacts quickly. Within 100 ms, tail latency observed by clients returns to normal.

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 keys, and leveraging load-aware scheduling to avoid overloading the servers. No change in 99% end-to-end latency is observed.

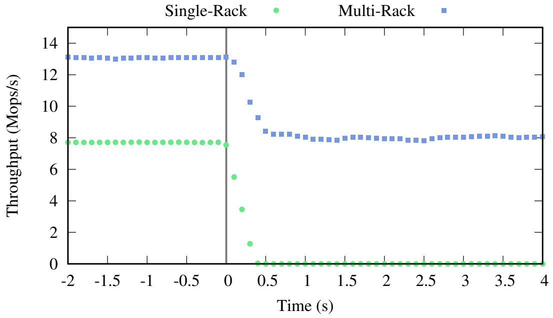


Figure 14: Throughput of single-rack vs. multi-rack configuration during a rack failure. After the failure ($t = 0$), the multi-rack configuration continues processing requests but loses some capacity.

9.8 Multi-Rack

To test a multi-rack configuration, we use a larger (but slower) cluster with 72 servers with dual 1.8 GHz Intel Xeon E5-2450 processors. These are organized into two racks, each with 24 storage servers and one Pegasus switch, plus a third rack of client machines. Per-node performance is significantly lower, largely because these servers use 10 Gbit NICs that do not support DPDK.

The two 24-server racks are configured into a 2-replica configuration: each rack acts as the head of the chain for half of the keys and the tail for the other half. Because both replicas need to handle WRITEs but only the tail processes READs, adding a second rack not only provides fault tolerance, it doubles read throughput; write throughput remains unchanged.

Figure 14 demonstrates this by comparing a single-rack and two-rack configurations, running a read-only workload with Zipf $\alpha = 1.2$; the two-rack configuration has 1.7× the throughput. At $t = 0$, one rack fails. The two-rack deployment is able to continue processing at half of its speed using the remaining rack. The single-rack deployment, of course, becomes entirely unavailable.

10 Related Work

Load Balancing. Load imbalance in large-scale key-value stores has been addressed by past systems in three ways. Consistent hashing [30] and virtual nodes [12] are widely used, but do not perform well with changing workloads. Solutions based on migration [11, 32, 61] and randomness [49] can be used to balance dynamic workloads, but these techniques introduce additional overheads and have limited ability to handle high skew. EC-Cache [57] balances load using erasure coding to split and replicate values, but works best for large keys in data-intensive clusters. SwitchKV [41] 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 [29] 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 [54], Beamer [52], and Maglev [16]; or using switches, as in SilkRoad [47] or Duet [20]. These systems are designed to balance long-lived flows across servers, whereas Pegasus balances load of individual request packets. Prism [23] provides a way to perform request-level load balancing by migrating TCP and TLS connections, an approach that could be useful for Pegasus as an alternative to its UDP-based protocol.

Several new systems use programmable switches for application-specific load balancing protocols. R2P2 [33] load balances RPCs for stateless services where any request can be handled by any server. Harmonia [69] allows optimized forwarding for read requests in replicated systems by tracking when concurrent writes are in progress.

Directory-Based Coherence. Directory-based coherence protocols have been used in a variety of shared-memory multiprocessors and distributed shared memory systems [4, 19, 22, 31, 34, 36, 37, 40]. 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 [44] 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 is involved in writes and coherence operations, limiting load-balancing opportunities for write-intensive workloads. Pegasus stores keys and values only in servers, and its coherence protocol allows any storage server to handle write requests, so Pegasus can load-balance both read- and write-intensive workloads. Both systems can scale beyond a rack and tolerate failures: IncBricks does so at the individual server level; Pegasus does so at the rack level.

11 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 10× 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.

Acknowledgments

We thank our shepherd Simon Peter and the anonymous reviewers for their valuable feedback. This work was supported by NSF grant CNS-1615102 and gifts from Google and VMware. Jialin Li was supported by an MOE AcRF Tier 1 grant. Ellis Michael was supported by an IBM fellowship. Xin Jin was supported in part by NSF grants CNS-1813487, CCF-1918757 and CNS-1955487, a Facebook Communications & Networking Research Award, and a Google Faculty Research Award.

References

[1] A. Adya, R. Grandl, D. Myers, and H. Qin. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS '19)*, Bertinoro, Italy, May 2019.

[2] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, Nov. 2016.

[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)*, 1990.

[5] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, Banff, AL, Canada, Nov. 2020.

[6] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, 2010.

[7] 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.

[8] 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.

[9] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*, 2011.

[10] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li. HotRing: A hotspot-aware in-memory key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, USA, Feb. 2020.

[11] Y. Cheng, A. Gupta, and A. R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*, Bordeaux, France, 2015.

[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, AL, Canada, 2001.

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

[14] 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.

[15] Intel Data Plane Development Kit. `https://www.dpdk.org/`.

[16] 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)*, Santa Clara, CA, 2016.

[17] 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.

[18] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the 12th ACM SIGOPS EuroSys (EuroSys '17)*, Belgrade, Serbia, Apr. 2017.

[19] 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.

[20] 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 SIGCOMM*, Chicago, IL, USA, 2014.

[21] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan. Scale-out ccNUMA: Exploiting skew with strongly consistent caching. In *Proceedings of the 13th European Conference on Systems (Eurosys '18)*, 2018.

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

[23] Y. Hayakawa, M. Honda, D. Santry, and L. Eggert. Prism: Proxies without the pain. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*, Boston, MA, USA, Feb. 2021.

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

[25] 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.

[26] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets '14)*, 2014.

[27] 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.

[28] Incomplete high-level spec prevents verification of exactly-once semantic. `https://github.com/microsoft/Ironclad/issues/3`.

[29] 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.

[30] 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 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, El Paso, Texas, USA, 1997.

[31] 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*, WTEC '94, San Francisco, CA, USA, 1994.

[32] 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)*, 2012.

[33] M. Koglas, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference*, Renton, WA, USA, July 2019.

[34] 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)*, 1994.

[35] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), Dec. 2001.

[36] 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)*, 1997.

[37] 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)*, Seattle, WA, USA, 1990.

[38] 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.

[39] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, 2016.

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

[41] 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, USA, 2016.

[42] 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.

[43] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

[44] 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 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, Xi'an, China, 2017.

[45] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. DistCache: Provable load balancing for large-scale storage systems with distributed caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, USA, Feb. 2019. USENIX.

[46] Y. Mao, E. Kohler, and R. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*, Bern, Switzerland, Apr. 2012.

[47] 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 2017 ACM SIGCOMM*, Los Angeles, CA, USA, 2017.

[48] 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.

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

[50] 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, USA, 2013.

[51] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. An analysis of load imbalance in scale-out data serving. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS '16)*, 2016.

[52] 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)*, Renton, WA, 2018.

[53] 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.

[54] 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 2013 ACM SIGCOMM*, SIGCOMM '13, Hong Kong, China, 2013.

[55] Pegasus public repository. `https://github.com/NUS-Systems-Lab/pegasus`.

[56] D. R. K. Ports and J. Nelson. When should the network be the computer? In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS '19)*, Bertinoro, Italy, May 2019.

[57] 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, Savannah, GA, USA, 2016.

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

[59] 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.

[60] 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.

[61] 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.

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

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

[64] Barefoot Tofino 2: Second-generation of world's fastest P4-programmable Ethernet switch ASICs. `https://www.barefootnetworks.com/products/brief-tofino-2/`.

[65] Broadcom's Tomahawk 3 Ethernet switch chip.

[66] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI '04)*, San Francisco, CA, USA, 2004.

[67] J. Yang, Y. Yue, and R. Vinayak. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, Banff, AL, Canada, Nov. 2020.

[68] 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.

[69] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proceedings of the VLDB Endowment*, 13(3):376–389, Nov. 2019.

A Artifact Appendix

Abstract

Our artifact includes the following components: 1) P4 source code of the Pegasus switch data plane, 2) Python source code of the Pegasus switch controller, 3) C++ implementation of an in-memory key-value store with Intel DPDK, 4) configuration files and Python/shell scripts for running Pegasus experiments in a cluster, and 5) a TLA+ specification of the Pegasus protocol. The artifact is publicly available at: `https://github.com/NUS-Systems-Lab/pegasus`.

A.1 Artifact check-list

- **Algorithm:** Coherence protocol.
- **Program:** Key-value store, P4 packet processing program.
- **Compilation:** GCC 7.5.0 (Ubuntu 7.5.0-3ubuntu11̃8.04), Barefoot SDE 9.1.1
- **Binary:** Generated from GCC compiler and Barefoot SDE.
- **Run-time environment:** Ubuntu 18.04 LTS (Linux 4.15), Barefoot SDE 9.1.1
- **Hardware:** Dual socket 2.2 GHz Intel Xeon Silver 4114 processors with 20 cores and 48 GB RAM per socket. Mellanox ConnectX-4 25 Gbit NICs. Arista 7170-64S (barefoot Tofino-based) programmable switch.
- **Execution:** Bash and Python scripts.
- **Output:** Throughput. Average, median, 90%, 99% latencies.
- **Experiments:** Experiments as specified in the main paper (§9). Customizable experiment parameters: number of clients and servers, client request rate, read/write ratio, Zipfian coefficient, value size, number of keys, maximum number of replicated objects, and experiment duration.
- **Expected experiment run time:** 10-60 seconds per experiment.
- **Public link:** `https://github.com/NUS-Systems-Lab/pegasus`
- **Code licenses:** MIT license.

A.2 Description

A.2.1 How to access

All source code, configuration files, and scripts are publicly available at: `https://github.com/NUS-Systems-Lab/pegasus`.

A.2.2 Hardware dependencies

The artifact requires a P4 programmable switch (e.g., Barefoot Tofino programmable switch ASIC). The network interface cards on the client and server machines need to support Intel DPDK.

A.2.3 Software dependencies

The artifact has been tested on Ubuntu 18.04 LTS (Linux kernel 4.15). Compiling and running the Pegasus P4 data plane program require the Barefoot SDE (tested with version 9.1.1). Additional software package dependencies:

- libevent
- Intel TBB
- libnuma
- zlib
- DPDK (tested with version 19.11)
- Python Sorted Containers
- Python PyREM

A.2.4 Data sets

Experiments in this artifact expect a text file that contains ASCII keys (one key per line) for the key-value store. We provide a sample keys file, `artifact_eval/keys`, that has one million 64B-keys.

A.3 Installation

First, download or clone the repository. Throughout this document, we will use the following macros:

- `$REPO`: path to the root of the repository
- `$SDE`: path to Barefoot SDE
- `$SDE_INSTALL`: path to Barefoot SDE installation directory

A.3.1 Compiling Client and Server Code

Run `make` in `$REPO`.

A.3.2 Compiling P4 Code

On the target P4 switch:

```
cd $SDE/pkgsrc/p4-build
./configure P4_PATH=$REPO/p4/p4_tofino/pegasus.p4 \
            P4_NAME=pegasus P4_PREFIX=pegasus \
            P4_VERSION=p4-14 P4_FLAGS="--verbose 2" \
            --with-tofino --prefix=$SDE_INSTALL \
            --enable-thrift
make && make install
./configure P4_PATH=$REPO/p4/netcache/one.p4 \
            P4_NAME=netcache P4_PREFIX=netcache \
            P4_VERSION=p4-14 P4_FLAGS="--verbose 2" \
            --with-tofino --prefix=$SDE_INSTALL \
            --enable-thrift
make && make install
```

Note that the location of `p4-build` may depend on the Barefoot SDE version.

A.4 Experiment workflow

A.4.1 P4 Switch

First, start the Pegasus switch daemon on the P4 switch:

```
cd $SDE
./run_switchd.sh -p pegasus
```

Or if running NetCache, run the following:

```
cd $SDE
./run_switchd.sh -p netcache
```

In the switch shell, add and enable all switch ports used by the experiments.

Secondly, modify `$REPO/artifact_eval/pegasus.json` and `$REPO/artifact_eval/netcache.json` with the testbed cluster configuration (refer to `artifact_eval/README.md` for configuration file format).

Thirdly, start the Pegasus switch controller:

```
cd $REPO
./artifact_eval/run_pegasus_controller.sh
```

Or if running NetCache, run the following:

```
cd $REPO
./artifact_eval/run_netcache_controller.sh
```

A.4.2 End-Hosts

First, modify `$REPO/artifact_eval/testbed.config` with the cluster configuration. Refer to `artifact_eval/README.md` for the format of the file.

Secondly, modify the experiment python script `$REPO/artifact_eval/run_experiments.py`. Update `clients` and `servers` with actual host names of the client and server machines.

Lastly, on a machine that has ssh connectivity to all clients and servers, run the following:

```
python2 $REPO/artifact_eval/run_experiments.py
```

A.5 Evaluation and expected result

The experiment python script outputs the following statistics:

- Total throughput
- Average latency
- Median latency
- 90% latency
- 99% latency

Modify `n_client_threads` and `interval` in the experiment script to control the client load. Tune them until getting the maximum throughput with some 99% latency SLO, as reported in the paper.

To evaluate the different workloads and system configurations as specified in §9, vary the following parameters in the experiment script:

- `n_servers`: number of servers used in the experiment
- `node_config`: one of `pegasus`, `netcache`, or `static` (consistent hashing). Note that `pegasus` and `netcache` require running the corresponding P4 switch daemon and controller.
- `alpha`: Zipfian coefficient
- `get_ratio`: percentage of read requests in the workload (0.0 - 1.0)
- `key_type`: key access distribution. Either `unif` (uniform) or `zipf` (Zipfian)
- `value_len`: value size (in bytes)
- `n_keys`: total number of keys

A.6 AE Methodology

Submission, reviewing and badging methodology:

- `https://www.usenix.org/conference/osdi20/call-for-artifacts`