DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching

Zaoxing Liu[*], *Zhihao Bai*[*], *Zhenming Liu*[†], *Xiaozhou Li*[◁],
Changhoon Kim[‡], *Vladimir Braverman*[*], *Xin Jin*[*], *Ion Stoica*[◇]

[*]*Johns Hopkins University* [†]*College of William and Mary* [◁]*Celer Network* [‡]*Barefoot Networks* [◇]*UC Berkeley*

Abstract

Load balancing is critical for distributed storage to meet strict service-level objectives (SLOs). It has been shown that a fast cache can guarantee load balancing for a clustered storage system. However, when the system scales out to multiple clusters, the fast cache itself would become the bottleneck. Traditional mechanisms like cache partition and cache replication either result in load imbalance between cache nodes or have high overhead for cache coherence.

We present DistCache, a new distributed caching mechanism that provides provable load balancing for large-scale storage systems. DistCache co-designs cache allocation with cache topology and query routing. The key idea is to partition the hot objects with independent hash functions between cache nodes in different layers, and to adaptively route queries with the power-of-two-choices. We prove that DistCache enables the cache throughput to increase linearly with the number of cache nodes, by unifying techniques from expander graphs, network flows, and queuing theory. DistCache is a general solution that can be applied to many storage systems. We demonstrate the benefits of DistCache by providing the design, implementation, and evaluation of the use case for emerging switch-based caching.

1 Introduction

Modern planetary-scale Internet services (e.g., search, social networking and e-commerce) are powered by large-scale storage systems that span hundreds to thousands of servers across tens to hundreds of racks [1–4]. To ensure satisfactory user experience, the storage systems are expected to meet strict service-level objectives (SLOs), regardless of the workload distribution. A key challenge for scaling out is to achieve load balancing. Because real-world workloads are usually highly-skewed [5–8], some nodes receive more queries than others, causing hot spots and load imbalance. The system is bottle-necked by the overloaded nodes, resulting in low throughput and long tail latencies.

Caching is a common mechanism to achieve load balancing [9–11]. An attractive property of caching is that caching $O(n \log n)$ hottest objects is enough to balance n storage nodes, regardless of the query distribution [9]. The cache size only relates to the number of storage nodes, despite the number of objects stored in the system. Such property leads to recent advancements like SwitchKV [10] and NetCache [11] for balancing clustered key-value stores.

Unfortunately, the small cache solution cannot scale out to multiple clusters. Using one cache node per cluster only provides *intra-cluster* load balancing, but not *inter-cluster* load balancing. For a large-scale storage system across many clusters, the load between clusters (where each cluster can be treated as one "big server") would be imbalanced. Using another cache node, however, is not sufficient, because the caching mechanism requires the cache to process *all* queries to the $O(n \log n)$ hottest objects [9]. In other words, the cache throughput needs to be no smaller than the *aggregate* throughput of the storage nodes.

As such, it requires another caching layer with multiple cache nodes for inter-cluster load balancing. The challenge is on cache allocation. Naively replicating hot objects to all cache nodes incurs high overhead for cache coherence. On the other hand, simply partitioning hot objects between the cache nodes would cause the load to be imbalanced between the cache nodes. The system throughput would still be bottle-necked by one cache node under highly-skewed workloads. Thus, the key is to carefully partition and replicate hot objects, in order to avoid load imbalance between the cache nodes, and to reduce the overhead for cache coherence.

We present DistCache, a new distributed caching mechanism that provides provable load balancing for large-scale storage systems. DistCache enables a "one big cache" abstraction, i.e., an *ensemble* of fast cache nodes acts as a single ultra-fast cache. DistCache co-designs cache allocation with multi-layer cache topology and query routing. The key idea is to use independent hash functions to partition hot objects between the cache nodes in different layers, and to apply the power-of-two-choices [12] to adaptively route queries.

Using independent hash functions for cache partitioning ensures that if a cache node is overloaded in one layer, then

the set of hot objects in this node would be distributed to multiple cache nodes in another layer with high probability. This intuition is backed up by a rigorous analysis that leverages expander graphs and network flows, i.e., we prove that there exists a solution to split queries between different layers so that no cache node would be overloaded in any layer. Further, since a hot object is only replicated in each layer once, it incurs minimal overhead for cache coherence.

Using the power-of-two-choices for query routing provides an efficient, distributed, online solution to split the queries between the layers. The queries are routed to the cache nodes in a distributed way based on cache loads, without central coordination and without knowing what is the optimal solution for query splitting upfront. We leverage queuing theory to show it is asymptotically optimal. The major difference between our problem and the balls-and-bins problem in the original power-of-two-choices algorithm [12] is that our problem hashes objects into cache nodes, and queries to the same object *reuse* the same hash functions to choose hash nodes, instead of using a *new random source* to sample two nodes for each query. We show that the power-of-two-choices makes a "life-or-death" improvement in our problem, instead of a "shaving off a log n" improvement.

DistCache is a general caching mechanism that can be applied to many storage systems, e.g., in-memory caching for SSD-based storage like SwitchKV [10] and switch-based caching for in-memory storage like NetCache [11]. We provide a concrete system design to scale out NetCache to demonstrate the power of DistCache. We design both the control and data planes to realize DistCache for the emerging switch-based caching. The controller is highly scalable as it is off the critical path. It is only responsible for computing the cache partitions and is not involved in handling queries. Each cache switch has a local agent that manages the hot objects of its own partition.

The data plane design exploits the capability of programmable switches, and makes innovative use of *in-network telemetry* beyond traditional network monitoring to realize *application-level functionalities*—disseminating the loads of cache switches by piggybacking in packet headers, in order to aid the power-of-two-choices. We apply a two-phase update protocol to ensure cache coherence.

In summary, we make the following contributions.

- We design and analyze DistCache, a new distributed caching mechanism that provides provable load balancing for large-scale storage systems (§3).
- We apply DistCache to a use case of emerging switch-based caching, and design a concrete system to scale out an in-memory storage rack to multiple racks (§4).
- We implement a prototype with Barefoot Tofino switches and commodity servers, and integrate it with Redis (§5). Experimental results show that DistCache scales out linearly with the number of racks, and the cache coherence protocol incurs minimal overhead (§6).

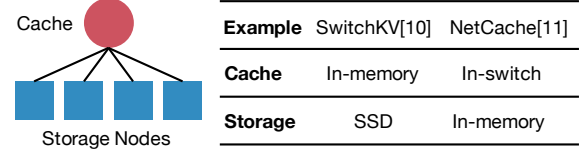|	Example	SwitchKV[10]	NetCache[11]
Cache		In-memory	In-switch
Storage		SSD	In-memory

Figure 1: Background on caching. If the cache node can absorb *all* queries to the hottest $O(n \log n)$ objects, the load on the storage nodes is guaranteed to be balanced [9].

2 Background and Motivation

2.1 Small, Fast Cache for Load Balancing

As a building block of Internet applications, it is critical for storage systems to meet strict SLOs. Ideally, given the per-node throughput T, a storage system with n nodes should guarantee a total throughput of $n \cdot T$. However, real-world workloads are usually high-skewed, making it challenging to guarantee performance [5–8]. For example, a measurement study on the Memcached deployment shows that about 60-90% of queries go to the hottest 10% objects [5].

Caching is a common mechanism to achieve load balancing for distributed storage, as illustrated in Figure 1. Previous work has proven that if the cache node can absorb *all* queries to the hottest $O(n \log n)$ objects, then the load on n storage servers is guaranteed to be balanced, despite query distribution and the total number of objects [9]. However, it also requires that the cache throughput needs to be at least $n \cdot T$ to not become the system bottleneck. Based on this theoretical foundation, SwitchKV [10] uses an in-memory cache to balance SSD-based storage nodes, and NetCache [11] uses a switch-based cache to balance in-memory storage nodes. Empirically, these systems have shown that caching a few thousand objects is enough for balancing a hundred storage nodes, even for highly-skewed workloads like Zipfian-0.9 and Zipfian-0.99 [10, 11].

2.2 Scaling out Distributed Storage

The requirement on the cache performance limits the system scale. Suppose the throughput of a cache node is $\widetilde{T} = c \cdot T$. The system can scale to at most a cluster of c storage nodes. For example, given that the typical throughput of a switch is 10-100 times of that of a server, NetCache [11] can only guarantee load balancing for 10-100 storage servers. As such, existing solutions like SwitchKV [10] and NetCache [11] are constrained to one storage cluster, which is typically one or two racks of servers.

For a cloud-scale distributed storage system that spans many clusters, the load between the clusters can become imbalanced, as shown in Figure 2(a). Naively, we can put another cache node in front of all clusters to balance the load between clusters. At first glance, this seems a nice solution, since we can first use a cache node in each cluster for *intra-cluster* load balancing, and then use an upper-layer cache node for *inter-cluster* load balancing. However, now each cluster becomes a "big server", of which the throughput is already $\widetilde{T}$. Using
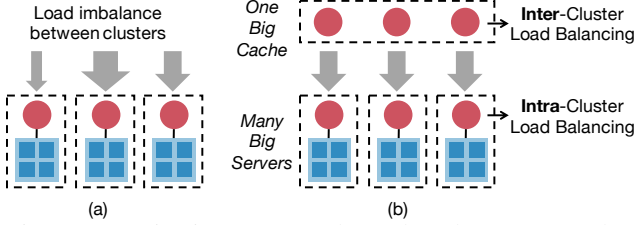
Figure 2: Motivation. (a) A cache node only guarantees load balancing for its own cluster, but the load between clusters can be unbalanced. (b) Use one cache node in each cluster for *intra-cluster* load balancing, and another layer of cache nodes for *inter-cluster* load balancing. The challenge is on cache allocation.

only one cache node cannot meet the cache throughput requirement, which is $m\widetilde{T}$ for m clusters. While using multiple upper-layer cache nodes like Figure 2(b) can potentially meet this requirement, it brings the question of how to allocate hot objects to the upper-layer cache nodes. We examine two traditional cache allocation mechanisms.

Cache partition. A straightforward solution is to partition the object space between the upper-layer cache nodes. Each cache node only caches the hot objects of its own partition. This works well for uniform workloads, as the cache throughput can grow linearly with the number of cache nodes. But remember that under uniform workloads, the load on the storage nodes is already balanced, obviating the need for caching in the first place. The whole purpose of caching is to guarantee load balancing for skewed workloads. Unfortunately, cache partition would cause load imbalance between the upper-layer cache nodes, because multiple hot objects can be partitioned to the same upper-layer cache node, making one cache node become the system bottleneck.

Cache replication. Cache replication replicates the hot objects to all the upper-layer cache nodes, and the queries can be uniformly sent to them. As such, cache replication can ensure that the load between the cache nodes is balanced, and the cache throughput can grow linearly with the number of cache nodes. However, cache replication introduces high overhead for cache coherence. When there is a write query to a cached object, the system needs to update both the primary copy at the storage node and the cached copies at the cache nodes, which often requires an expensive two-phase update protocol for cache coherence. As compared to cache partition which only caches a hot object in one upper-layer cache node, cache replication needs to update all the upper-layer cache nodes for cache coherence.

Challenge. Cache partition has low overhead for cache coherence, but cannot increase the cache throughput linearly with the number of cache nodes; cache replication achieves the opposite. Therefore, the main challenge is to carefully partition and replicate the hot objects, in order to (*i*) avoid load imbalance between upper-layer cache nodes, and to (*ii*)
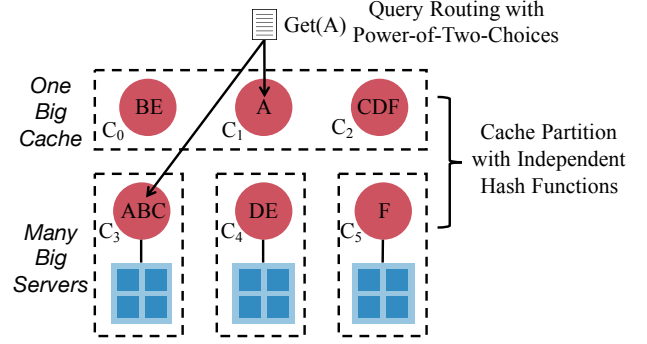

Figure 3: Key idea. (*i*) Use independent hash functions to partition hot objects in different layers. (*ii*) Use the power-of-two-choices to route queries, e.g., route $Get(A)$ to either cache node C_1 or cache node C_3 based on cache load.

reduce the overhead for cache coherence.

3 DistCache Caching Mechanism Design

3.1 Key Idea

We design DistCache, a new distributed caching mechanism to address the challenge described in §2.2. As illustrated by Figure 3, our key idea is to use independent hash functions for cache allocation and the power-of-two-choices for query routing, in order to balance the load between cache nodes. Our mechanism only caches an object at most once in a layer, incurring minimal overhead for cache coherence. We first describe the mechanism and the intuitions, and then show why it works in §3.2.

Cache allocation with independent hash functions. Our mechanism partitions the object space with independent hash functions in different layers. The lower-layer cache nodes primarily guarantee intra-cluster load balancing, each of which only caches hot objects for its own cluster, and thus each cluster appears as one "big server". The upper-layer cache nodes are primarily for inter-cluster load balancing, and use a different hash function for partitioning. The intuition is that if one cache node in a layer is overloaded by receiving too many queries to its cached objects, because the hash functions of the two layers are independent, the set of hot objects would be distributed to multiple cache nodes in another layer with high probability. Figure 3 shows an example. While cache node C_3 in the lower layer is overloaded with three hot objects (A, B and C), the three objects are distributed to three cache nodes (C_0, C_1 and C_2) in the upper layer. The upper-layer cache nodes only need to absorb queries for objects (e.g., A and B) that cause the imbalance *between* the clusters, and do not need to process queries for objects (e.g., D and F) that already *spread out* in the lower-layer cache nodes.

Query routing with the power-of-two-choices. The cache allocation strategy only tells that there exists a way to handle queries without overloading any cache nodes, but it does not tell how the queries should be *split* between the layers.

Conceivably, we could use a controller to collect global measurement statistics to infer the query distribution. Then the controller can compute an optimal solution and enforce it at the senders. Such an approach has high system complexity, and the responsiveness to dynamic workloads depends on the agility of the control loop.

Our mechanism uses an *efficient, distributed, online* solution based on the power-of-two-choices [12] to route queries. Specifically, the sender of a query only needs to look at the loads of the cache nodes that cache the queried object, and sends the query to the less-loaded node. For example, the query $Get(A)$ in Figure 3 is routed to either C_1 or C_3 based on their loads. The key advantage of our solution is that: it is distributed, so that it does not require a centralized controller or any coordination between senders; it is online, so that it does not require a controller to measure the query distribution and compute the solution, and the senders do not need to know the solution upfront; it is efficient, so that its performance is close to the optimal solution computed by a controller with perfect global information (as shown in §3.2). Queries to hit a lower-layer cache node can either pass through an arbitrary upper-layer node, or totally bypass the upper-layer cache nodes, depending on the actual use case, which we describe in §3.4.

Cache size and multi-layer hierarchical caching. Suppose there are m clusters and each cluster has l servers. First, we let each lower-layer cache node cache $O(l \log l)$ objects for its own cluster for *intra-cluster* load balancing, so that a total of $O(ml \log l)$ objects are cached in the lower layer and each cluster appears like one "big server". Then for *inter-cluster* load balancing, the upper-layer cache nodes only need to cache a total of $O(m \log m)$ objects. This is different from a single ultra-fast cache at a front-end that handles all ml servers directly. In that case, $O(ml \log(ml))$ objects need to be cached based on the result in [9]. However, in DistCache, we have an extra upper-layer (with the same total throughput as ml servers) to "refine" the query distribution that goes to the lower-layer, which reduces the effective cache size in the lower layer to $O(ml \log l)$. Thus, this is not a contradiction with the result in [9]. While these $O(m \log m)$ inter-cluster hot objects also need to be cached in the lower layer to enable the power-of-two-choices, most of them are also hot inside the clusters and thus have already been contained in the $O(ml \log l)$ intra-cluster hot objects.

Our mechanism can be applied recursively for multi-layer hierarchical caching. Specifically, applying the mechanism to layer i can balance the load for a set of "big servers" in layer i-1. Query routing uses the *power-of-k-choices* for k *layers*. Note that using more layers actually increases the total number of cache nodes, since each layer needs to provide a total throughput at least equal to that of all storage nodes. The benefit of doing so is on reducing the cache size. When the number of clusters is no more than a few hundred, a cache node has enough memory with two layers.

3.2 Analysis

Prior work [9] has shown that caching $O(n \log n)$ hottest objects in a single cache node can balance the load for n storage nodes for any query distribution. In our work, we replace the single cache node with multiple cache nodes in two layers to support a larger scale. Therefore, based on our argument on the cache size in §3.1, we need to prove that the two-layer cache can absorb all queries to the hottest $O(m \log m)$ objects under any query distribution for all m clusters. We first define a mathematical model to formalize this problem.

System model. There are k hot objects $\{o_0, o_1, \ldots, o_{k-1}\}$ with query distribution $P = \{p_0, p_1, \ldots, p_{k-1}\}$, where p_i denotes the fraction of queries for object o_i, and $\sum_{i=0}^{k-1} p_i = 1$. The total query rate is R, and the query rate for object o_i is $r_i = p_i \cdot R$. There are in total $2m$ cache nodes that are organized to two groups $A = \{a_0, a_1, \ldots, a_{m-1}\}$ and $B = \{b_0, b_1, \ldots, b_{m-1}\}$, which represent the upper and lower layers, respectively. The throughput of each cache node is $\widetilde{T}$.

The objects are mapped to the cache nodes with two independent hash functions $h_0(x)$ and $h_1(x)$. Object o_i is cached in a_{j_0} in group A and b_{j_1} in group B, where $j_0 = h_0(i)$ and $j_1 = h_1(i)$. A query to o_i can be served by either a_{j_0} or b_{j_1}.

Goal. Our goal is to evaluate the total query rate R the cache nodes can support, in terms of m and $\widetilde{T}$, regardless of query distribution P, as well as the relationship between k and m. Ideally, we would like $R \approx \alpha m \widetilde{T}$ where α is a small constant (e.g., 1), so that the operator can easily provision the cache nodes to meet the cache throughput requirement (i.e., no smaller than the total throughput of storage nodes).

If we can set k to be $O(m \log m)$, it means that the cache nodes can absorb all queries to the hottest $O(m \log m)$ objects, despite query distribution. Combining this result with the cache size argument in §3.1, we can prove that the distributed caching mechanism can provide performance guarantees for large-scale storage systems across multiple clusters.

A perfect matching problem in a bipartite graph. The key observation of our analysis is that the problem can be converted to finding a perfect matching in a bipartite graph. Intuitively, if a perfect matching exists, the requests to k hot objects can be completely absorbed from the two layers of cache nodes. Specifically, we construct a bipartite graph $G = (U, V, E)$, where U is the set of vertices on the left, V is the set of vertices on the right, and E is the set of edges. Let U represent the set of objects, i.e., $U = \{o_0, o_1, \ldots, o_{k-1}\}$. Let V represent the set of cache nodes, i.e., $V = A \cup B = \{a_0, a_1, \ldots, a_{m-1}, b_0, b_1, \ldots, b_{m-1}\}$. Let E represent the hash functions mapping from the objects to the cache nodes, i.e., $E = \{e_{o_i, a_{j_0}} | h_0(i) = j_0\} \cup \{e_{o_i, b_{j_1}} | h_1(i) = j_1\}$. Given a query distribution P and a total query rate R, we define a perfect matching in G to represent that the workload can be supported by the cache nodes.

Definition 1. *Let $\Gamma(v)$ be the set of neighbors of vertex v in G.*
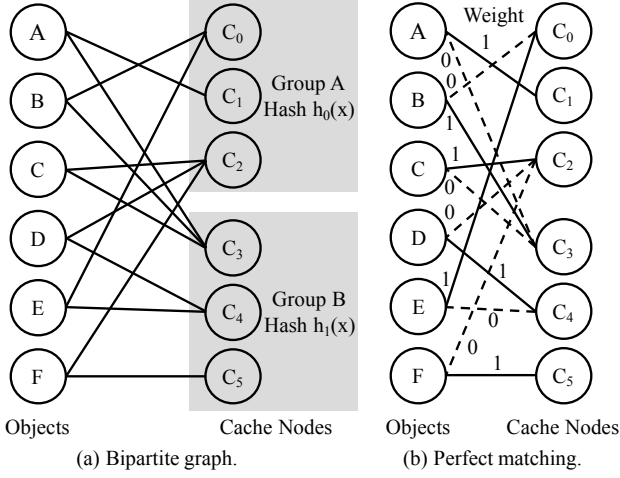
Figure 4: Example for analysis. (a) A bipartite graph constructed for the scenario in Figure 3. (b) A perfect matching for query routing when all objects have a query rate of 1, and all cache nodes have a throughput of 1.

A weight assignment $W = \{w_{i,j} \in [0, \widetilde{T}] | e_{i,j} \in E\}$ is a perfect matching of G if

1. $\forall o_i \in U : \quad \sum_{v \in \Gamma(o_i)} w_{o_i,v} = p_i \cdot R$, and
2. $\forall v \in V : \quad \sum_{u \in \Gamma(v)} w_{u,v} \leq \widetilde{T}$.

In this definition, $w_{i,j}$ denotes the portion of the queries to object i served by cache node j. Condition 1 ensures that for any object o_i, its query rate $p_i \cdot R$ is fully served. Condition 2 ensures that for any cache node v, its load is no more than $\widetilde{T}$, i.e., no single cache node is overloaded.

When a perfect matching exists, it is feasible to serve all the queries by the cache nodes. We use the example in Figure 4 to illustrate this. Figure 4(a) shows the bipartite graph constructed for the scenario in Figure 3, which contains six hot objects (*A-F*) and six cache nodes in two layers (C_0-C_5). The edges are built based on two hash functions $h_0(x)$ and $h_1(x)$. Figure 4(b) shows a perfect matching for the case that all objects have the same query rate $r_i = 1$ and all cache nodes have the same throughput $\widetilde{T} = 1$. The number besides an edge denotes the weight of an edge, i.e., the rate of the object served by the cache node. For instance, all queries to A are served by C_1. This is a simple example to illustrate the problem. In general, the query rates of the objects do not have to be the same, and the queries to one object may be served by multiple cache nodes.

Step 1: existence of a perfect matching. We first show the existence of a perfect matching for any given total rate R and *any* query distribution P. We have the following lemma to demonstrate how big the total rate R can be in terms of $\widetilde{T}$, for any P. For the full proof of Lemma 1, we refer the readers to §A.2 in the technical report [13].

Lemma 1. *Let α be a suitably small constant. If $k \leq m^\beta$ for some constant β (i.e., k and m are polynomial-related) and $\max_i(p_i) \cdot R \leq \widetilde{T}/2$, then for any $\varepsilon > 0$, there exists a*

perfect matching for $R = (1 - \varepsilon)\alpha \cdot m\widetilde{T}$ and any P, with high probability for sufficiently large m.

Proof sketch of Lemma 1: We utilize the results and techniques developed from expander graphs and network flows. (*i*) We first show that G has the so-called *expansion property* with high probability. Intuitively, the property states that the neighborhood of any subset of nodes in U *expands*, i.e., for any $S \subseteq U$, $|\Gamma(S)| \geq |S|$. It has been observed that such properties exist in a wide range of random graphs [14]. While our G behaves similar to random bipartite graphs, we need the expansion property to hold for S in any size, which is stricter than the standard definition (which assumes S is not too large) and thus requires more delicate probabilistic techniques. (*ii*) We then show that if a graph has the expansion property, then it has a perfect matching. This step can be viewed as a generalization of Hall's theorem [15] in our setting. Hall's theorem states that a balanced bipartite graph has a perfect (non-fractional) matching if and only if for any subset S of the left nodes, $|\Gamma(S)| \geq |S|$. and perfect matching can be fractional. This step can be proved by the max-flow-min-cut theorem, i.e., expansion implies large cut, and then implies large matching.

Step 2: finding a perfect matching. Demonstrating the existence of a perfect matching is insufficient since it just ensures the queries can be absorbed but does not give the actual weight assignment W, i.e., how the cache nodes should serve queries for each P to achieve R. This means that the system would require an algorithm to compute W and an mechanism to enforce W. As discussed in §3.1, instead of doing so, we use the power-of-two-choices to "emulate" the perfect matching, without the need to know what the perfect matching is. The quality of the mechanism is backed by Lemma 2, which we prove using queuing theory. The detailed proof can be found in §A.3 of the technical report [13].

Lemma 2. *If a perfect matching exists for G, then the power-of-two-choices process is stationary.*

Stationary means that the load on the cache nodes would converge, and the system is "sustainable" in the sense that the system will never "blow up" (i.e., build up queues in a cache node and eventually drop queries) with query rate R.

Proof sketch of Lemma 2: Showing this lemma requires us to use a powerful building block in query theory presented in [16, 17]. Consider $2m$ exponential random variables with rate $\widetilde{T}_i > 0$. Each non-empty set of cache nodes $S \subseteq [2m]$, has an associated Poisson arrival process with rate $\lambda_S \geq 0$ that joins the shortest queue in S with ties broken randomly. For each non-empty subset $Q \subseteq [2m]$, define the traffic intensity on Q as

$$\rho_Q = \frac{\sum_{S \subseteq Q} \lambda_S}{\mu_Q},$$

where $\mu_Q = \sum_{i \in Q} \widetilde{T}_i$. Note that the total rate at which objects served by Q can be greater than the numerator of (3.2) since other requests may be allowed to be served by some or all of

the cache nodes in Q. Let $\rho_{\max} = \max_{Q \subseteq [2m]} \{\rho_Q\}$. Given the result in [16, 17], if we can show $\rho_{\max} < 1$, then the Markov process is positive recurrent and has a stationary distribution. In fact, our cache querying can be described as an arrival process (§A.3 of [13]). Finally, we show that $\rho_{\max}$ is less than 1 and thus the process is stationary.

Step 3: main theorem. Based on Lemma 1 and Lemma 2, we can prove that our distributed caching mechanism is able to provide a performance guarantee, despite query distribution.

Theorem 1. *Let α be a suitable constant. If $k \leq m^\beta$ for some constant β (i.e., k and m are polynomial-related) and $\max_i(p_i) \cdot R \leq \widetilde{T}/2$, then for any $\varepsilon > 0$, the system is stationary for $R = (1-\varepsilon)\alpha \cdot m\widetilde{T}$ and any P, with high probability for sufficiently large m.*

Interpretation of the main theorem: As long as the query rate of a single hot object o_i is no larger than $\widetilde{T}/2$ (e.g., half of the entire throughput in a cluster rack), DistCache can support a query rate of $\approx m\widetilde{T}$ for any query distributions to the k hot objects (where k can be fairly large in terms of m) by using the power-of-two-choices protocol to route the queries to the cached objects. The key takeaways are presented in the following section.

3.3 Remarks

Our problem isn't a balls-in-bins problem using the original power-of-two-choices. The major difference is that our problem hashes objects into cache nodes, and queries to the same object by *reusing* the same hash functions, instead of using a *new random source* to sample two nodes for each query. In fact, without using the power-of-two-choices, the system is in *non-stationary*. This means that the power-of-two-choices makes a "life-or-death" improvement in our problem, instead of a "shaving off a log n" improvement. While we refer to the technical report [13] for detailed discussions, we have a few important remarks.

- **Nonuniform number of cache nodes in two layers.** For simplicity we use the same number of m cache nodes per layer in the system. However, we can generalize the analysis to accommodate the cases of different numbers of caches nodes in two layers, as long as $\min(m_0, m_1)$ is sufficiently large, where m_0 and m_1 are the number of upper-layer and lower-layer cache nodes respectively. While it requires m to be sufficiently large, it is not a strict requirement, because the load imbalance issue is only significant when m is large.
- **Nonuniform throughput of cache nodes in two groups.** Although our analysis assumes the throughput of a cache node is $\widetilde{T}$, we can generalize it to accommodate the cases of nonuniform throughput by treating a cache node with a large throughput as multiple smaller cache nodes with a small throughput.
- **Cache size.** As long as the number of objects and the number of cache nodes are polynomially-related ($k \leq m^\beta$), the

system is able to provide the performance guarantee. It is more relaxed than $O(m \log m)$. Therefore, by setting $k = O(m \log m)$, the cache nodes are able to absorb all queries to the hottest $O(m \log m)$ objects, making the load on the m clusters balanced.
- **Maximum query rate for one object.** The theorem requires that the maximum query rate for one object is no bigger than half the throughput of one cache node. This is not a severe restriction for the system, because a cache node is orders of magnitude faster than a storage node.
- **Performance guarantee.** The system can guarantee a total throughput of $R = (1-\varepsilon)\alpha \cdot m\widetilde{T}$, which scales linearly with m and T. In practice, α is close to 1.

3.4 Use Cases

DistCache is a general solution that can be applied to scale out various storage systems (e.g., key-value stores and file systems) using different storage mediums (e.g., HDD, SDD and DRAM). We describe two use cases.

Distributed in-memory caching. Based on the performance gap between DRAMs and SSDs, a fast in-memory cache node can be used balance an SSD-based storage cluster, such as SwitchKV [10]. DistCache can scale out SwitchKV by using another layer of in-memory cache nodes to balance multiple SwitchKV clusters. While it is true that multiple in-memory cache nodes can be balanced using a faster switch-based cache node, applying DistCache obviates the need to introduce a new component (i.e., a switch-based cache) to the system. Since the queries are routed to the cache and storage nodes by the network, queries to the lower-layer cache nodes can totally bypass the upper-layer cache nodes.

Distributed switch-based caching. Many low-latency storage systems for interactive web services use more expensive in-memory designs. An in-memory storage rack can be balanced by a switch-based cache like NetCache [11], which directly caches the hot objects in the data plane of the ToR switch. DistCache can scale out NetCache to multiple racks by caching hot objects in a higher layer of the network topology, e.g., the spine layer in a two-layer leaf-spine network. As discussed in the remarks (§3.3), DistCache accommodates the cases that the number of spine switches is smaller and each spine switch is faster. As for query routing, while queries to hit the leaf cache switches need to inevitably go through the spine switches, these queries can be arbitrarily routed through any spine switches, so that the load on the spine switches can be balanced.

Note that while existing solutions (e.g., NetCache [11]) directly embeds caching in the switches which may raise concerns on deployment, another option for easier deployment is to use the cache switches as *stand-alone specialized appliances* that are separated from the switches in the data-center network. DistCache can be applied to scale out these specialized switch-based caching appliances as well.
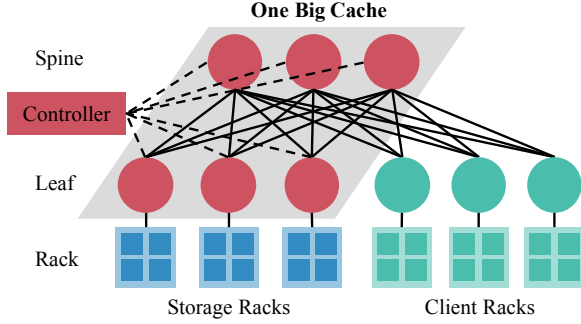
Figure 5: Architecture for distributed switch-based caching.

4 DistCache for Switch-Based Caching

To demonstrate the benefits of DistCache, we provide a concrete system design for the emerging switch-based caching. A similar design can be applied to other use cases as well.

4.1 System Architecture

Emerging switch-based caching, such as NetCache [11] is limited to one storage rack. We apply DistCache to switch-based caching to provide load balancing for cloud-scale key-value stores that span many racks. Figure 5 shows the architecture for a two-layer leaf-spine datacenter network.

Cache Controller. The controller computes the cache partitions, and notifies the cache switches. It updates the cache allocation under system reconfigurations, e.g., adding new racks and cache switches, and system failures; and thus updating the allocation is an infrequent task. We assume the controller is reliable by replicating on multiple servers with a consensus protocol such as Paxos [18]. The controller is not involved in handling storage queries in the data plane.

Cache switches. The cache switches provide two critical functionalities for DistCache: (1) caching hot key-value objects; (2) distributing switch load information for query routing. First, a local agent in the switch OS receives its cache partition from the controller, and manages the hot objects for its partition in the data plane. Second, the cache switches implement a lightweight in-network telemetry mechanism to distribute their load information by piggybacking in packet headers. The functionalities for DistCache are invoked by a reserved L4 port, so that DistCache does not affect other network functionalities. We use existing L2/L3 network protocols to route packets, and do not modify other network functionalities already in the switch.

ToR switches at client racks. The ToR switches at client racks provide query routing. It uses the power-of-two-choices to decide which cache switch to send a query to, and uses existing L2/L3 network protocols to route the query.

Storage servers. The storage servers host the key-value store. DistCache runs a shim layer in each storage server to integrate the in-network cache with existing key-value store software like Redis [19] and Memcached [20]. The shim layer also implements a cache coherence protocol to guarantee the consistency between the servers and cache switches.

Clients. DistCache provides a client library for applications to access the key-value store. The library provides an interface similar to existing key-value stores. It maps function calls from applications to DistCache query packets, and gathers DistCache reply packets to generate function returns.

4.2 Query Handling

A key advantage of DistCache is that it provides a distributed *on-path* cache to serve queries at line rate. Read queries on cached objects (i.e., cache hit) are directly replied by the cache switches, *without the need to visit storage servers*; read queries on uncached objects (i.e., cache miss) and write queries are forwarded to storage servers, *without any routing detour*. Further, while the cache is distributed, our query routing mechanism based on the power-of-two-choices ensures that the load between the cache switches is balanced.

Query routing at client ToR switches. Clients send queries via the client library, which simply translates function calls to query packets. The complexity of query routing is done at the ToR switches of the client racks. The ToR switches use the switch on-chip memory to store the loads of the cache switches. For each read query, they compare the loads of the switches that contain the queried object in their partitions, and pick the less-loaded cache switch for the query. After the cache switch is chosen, they use the existing routing mechanism to send the query to the cache switch. The routing mechanism can pick a routing path that balances the traffic in the network, which is orthogonal to this paper. Our prototype uses a mechanism similar to CONGA [21] and HULA [22] to choose the least loaded path to the cache switch.

For a cache hit, the cache switch copies the value from its on-chip memory to the packet, and returns the packet to the client. For a cache miss, the cache switch forwards the packet to the corresponding storage server that stores the queried object. Then the server processes the query and replies to the client. Figure 6 shows an example. A client in rack R_3 sends a query to read object A. Suppose A is cached in switch S_1 and S_3, and is stored in a server in rack R_0. The ToR switch S_6 uses the power-of-two-choices to decide whether to choose S_1 or S_3. Upon a cache hit, the cache switch (either S_1 or S_3) directly replies to the client (Figure 6(a)). Upon a cache miss, the query is sent to the server. But no matter whether the leaf cache (Figure 6(b)) or the spine cache (Figure 6(c)) is chosen, there is no routing detour for the query to reach R_0 after a cache miss.

Write queries are directly forwarded to the storage servers that contain the objects. The servers implement a cache coherence protocol for data consistency as described in §4.3.

Query processing at cache switches. Cache switches use the on-chip memory to cache objects in their own partitions. In programmable switches such as Barefoot Tofino [23], the on-
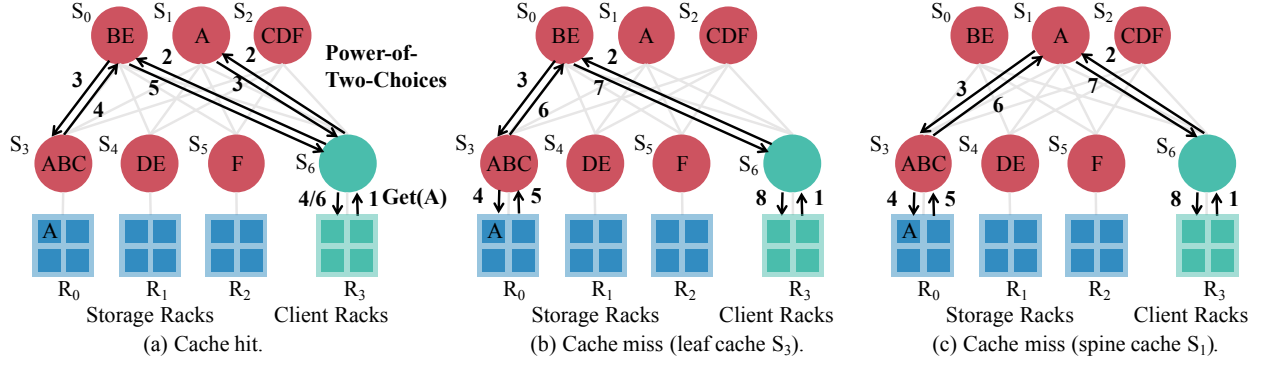
Figure 6: Query handling for $Get(A)$. S_6 uses the power-of-two-choices to decide whether to send $Get(A)$ to S_1 or S_3. (a) Upon a cache hit, the switch directly replies the query, without visiting the storage server. (b, c) Upon a cache miss, the query is forwarded to the storage server without routing detour.

chip memory is organized as register arrays spanning multiple stages in the packet processing pipeline. The packets can read and update the register arrays at line rate. We uses the same mechanism as NetCache [11] to implement a key-value cache that can support variable-length values, and a heavy-hitter (HH) detector that the switch local agent uses to decide what top k hottest objects in its partition to cache.

In-network telemetry for cache load distribution. We use a light-weight in-network telemetry mechanism to distribute the cache load information for query routing. The mechanism piggybacks the switch load (i.e., the total number of packets in the last second) in the packet headers of reply packets, and thus incurs minimal overhead. Specifically, when a reply packet of a query passes a cache switch, the cache switch adds its load to the packet header. Then when the reply packet reaches the ToR switch of the client rack, the ToR switch retrieves the load in the packet header to update the load stored in its on-chip memory. To handle the case that the cache load may become stale without enough traffic for piggybacking, we can add a simple aging mechanism that would gradually decrease a load to zero if the load is not updated for a long time. Note that aging is commonly supported by modern switch ASICs, but it is not supported by P4 yet, and thus is not implemented in our prototype.

4.3 Cache Coherence and Cache Update

Cache coherence. Cache coherence ensures data consistency between storage servers and cache switches when write queries update the values of the objects. The challenge is that an object may be cached in multiple cache switches, and need to be updated atomically. Directly updating the copies of an object in the cache switches may result in data inconsistency. This is because the cache switches are updated asynchronously, and during the update process, there would be a mix of old and new values at different switches, causing read queries to get different values from different switches.

We leverage the classic two-phase update protocol [24] to ensure strong consistency, where the first phase invalidates

all copies and the second phase updates all copies. To apply the protocol to our scenario, after receiving a write query, the storage server generates a packet to invalidate the copies in the cache switches. The packet traverses a path that includes all the switches that cache the object. The return of the invalidation packet indicates that all the copies are invalidated. Otherwise, the server resends the invalidation packet after a timeout. Figure 7(a) shows an example that the copies of object A are invalidated by an invalidation packet via path R_0-S_3-S_1-S_3-R_0. After the first phase, the server can update its primary copy, and send an acknowledgment to the client, instead of waiting for the second phase, as illustrated by Figure 7(b). This optimization is safe, since all copies are invalid. Finally, in the second phase, the server sends an update packet to update the values in the cache switches, as illustrated by Figure 7(c).

Cache update. The cache update is performed in a decentralized way without the involvement of the controller. We use a similar mechanism as NetCache [11]. Specifically, the local agent in each switch uses the HH detector in the data plane to detect hot objects in its own partition, and decides cache insertions and evictions. Cache evictions can be directly done by the agent; cache insertions require the agent to contact the storage servers. Slightly different from NetCache, DistCache uses a cleaner, more efficient mechanism to unify cache insertions and cache coherence. Specifically, the agent first inserts the new object into the cache, but marks it as invalid. Then the agent notifies the server; the server updates the cached object in the data plane using phase 2 of cache coherence, and serializes this operation with other write queries. As for comparison, in NetCache, the agent copies the value from the server to the switch via the switch control plane (which is slower than the data plane), and during the copying, the write queries to the object are blocked on the server.

4.4 Failure Handling

Controller failure. The controller is replicated on multiple servers for reliability (§4.1). Since the controller is only re-
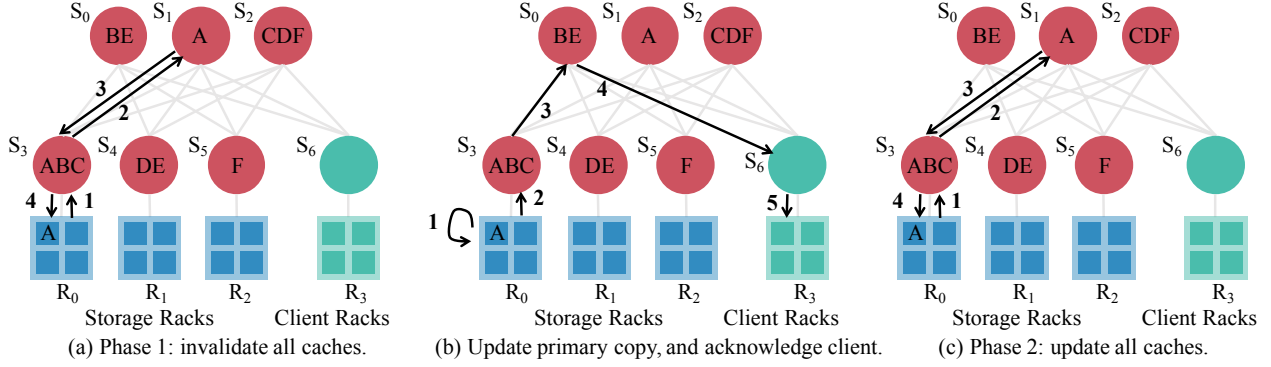
Figure 7: Cache coherence is achieved by a two-phase update protocol in DistCache. The example shows the process to handle an update to object A stored in rack R_0 with the two-phase update protocol.

sponsible for cache allocation, even if all servers of the controller fail, the data plane is still operational and hence processes queries. The servers can be simply rebooted.

Link failure. A link failure is handled by existing network protocols, and does not affect the system, as long as the network is connected and the routing is updated. If the network is partitioned after a link failure, the operator would choose between consistency and availability, as stated by the CAP theorem. If consistency were chosen, all writes should be blocked; if availability were chosen, queries can still be processed, but cache coherence cannot be guaranteed.

ToR switch failure. The servers in the rack would lose access to the network. The switch needs to be rebooted or replaced. If the switch is in a storage rack, the new switch starts with an empty cache and uses the cache update process to populate its cache. If the switch is in a client rack, the new switch initializes the loads of all cache switches to be zero, and uses the in-network telemetry mechanism to update them with reply packets.

Other Switch failure. If the switch is not a cache switch, the failure is directly handled by existing network protocols. If the switch is a cache switch, the system loses throughput provided by this switch. If it can be quickly restored (e.g., by rebooting), the system simply waits for the switch to come back online. Otherwise, the system remaps the cache partition of the failed switch to other switches, so that the hot objects in the failed switch can still be cached, alleviating the impact on the system throughput. The remapping leverages consistent hashing [25] and virtual nodes [26] to spread the load. Finally, if the network is partitioned due to a switch failure, the operator would choose consistency or availability, similar to that of a link failure.

5 Implementation

We have implemented a prototype of DistCache to realize distributed switch-based caching, including cache switches, client ToR switches, a controller, storage servers and clients.

Cache switch. The data plane of the cache switches is written

in the P4 language [27], which is a domain-specific language to program the packet forwarding pipelines of data plane devices. P4 can be used to program the switches that are based on Protocol Independent Switch Architecture (PISA). In this architecture, we can define the packet formats and packet processing behaviors by a series of match-action tables. These tables are allocated to different processing stages in a forwarding pipeline, based on hardware resources. Our implementation is compiled to Barefoot Tofino ASIC [23] with Barefoot P4 Studio software suite [28]. In the Barefoot Tofino switch, we implement a key-value cache module uses 16-byte keys, and contains 64K 16-byte slots per stage for 8 stages, providing values at the granularity of 16 bytes and up to 128 bytes without packet recirculation or mirroring. The Heavy Hitter detector module contains a Count-Min sketch [29], which has 4 register arrays and 64K 16-bit slots per array, and a Bloom filter, which has 3 register arrays and 256K 1-bit slots per array. The telemetry module uses one 32-bit register slot to store the switch load. We reset the counters in the HH detector and telemetry modules in every second. The local agent in the switch OS is written in Python. It receives cache partitions from the controller, and manages the switch ASIC via the switch driver using a Thrift API generated by the P4 compiler. The routing module uses standard L3 routing which forwards packets based on destination IP address.

Client ToR switch. The data plane of client ToR switches is also written in P4 [27] and is compiled to Barefoot Tofino ASIC [23]. Its query routing module contains a register array with 256 32-bit slots to store the load of cache switches. The routing module uses standard L3 routing, and picks the least loaded path similar to CONGA [21] and HULA [22].

Controller, storage server, and client. The controller is written in Python. It computes cache partitions and notifies the result to switch agents through Thrift API. The shim layer at each storage server implements the cache coherence protocol, and uses the hiredis library [30] to hook up with Redis [19]. The client library provides a simple key-value interface. We use the client library to generate queries with different distributions and different write ratios.

6 Evaluation

6.1 Methodology

Testbed. Our testbed consists of two 6.5Tbps Barefoot Tofino switches and two server machines. Each server machine is equipped with a 16 core-CPU (Intel Xeon E5-2630), 128 GB total memory (four Samsung 32GB DDR4-2133 memory), and an Intel XL710 40G NIC.

The goal is to apply DistCache to switch-based caching to provide load balancing for cloud-scale in-memory key-value stores. Because of the limited hardware resources we have, we are unable to evaluate DistCache at full scale with tens of switches and hundreds of servers. Nevertheless, we make the most of our testbed to evaluate DistCache by dividing switches and servers into multiple logical partitions and running real switch data plane and server software, as shown in Figure 8. Specifically, a physical switch emulates several virtual switches by using multiple queues and uses counters to rate limit each queue. We use one Barefoot Tofino switch to emulate the spine switches, and the other to emulate the leaf switches. Similarly, a physical server emulates several virtual servers by using multiple queues. We use one server to emulate the storage servers, and the other to emulate the clients. We would like to emphasize that the testbed runs the real switch data plane and runs the Redis key-value store [19] to process real key-value queries.

Performance metric. By using multiple processes and using the pipelining feature of Redis, our Redis server can achieve a throughput of 1 MQPS. We use Redis to demonstrate that DistCache can integrate with production-quality open-source software that is widely deployed in real-world systems. We allocate the 1 MQPS throughput to the emulated storage servers equally with rate limiting. Since a switch is able to process a few BQPS, the bottleneck of the testbed is on the Redis servers. Therefore, we use rate limiting to match the throughput of each emulated switch to the aggregated throughput of the emulated storage servers in a rack. We normalize the system throughput to the throughput of one emulated key-value server as the performance metric.

Workloads. We use both uniform and skewed workloads in the evaluation. The uniform workload generates queries to each object with the same probability. The skewed workload follows Zipf distribution with a skewness parameter (e.g., 0.9, 0.95, 0.99). Such skewed workload is commonly used to benchmark key-value stores [10, 31], and is backed by measurements from production systems [5, 6]. The clients use approximation techniques [10, 32] to quickly generate queries according to a Zipf distribution. We store a total of 100 million objects in the key-value store. We use Zipf-0.99 as the default query distribution to show that DistCache performs well even under extreme scenarios. We vary the skewness and the write ratio (i.e., the percentage of write queries) in the experiments to evaluate the performance of DistCache under different scenarios.
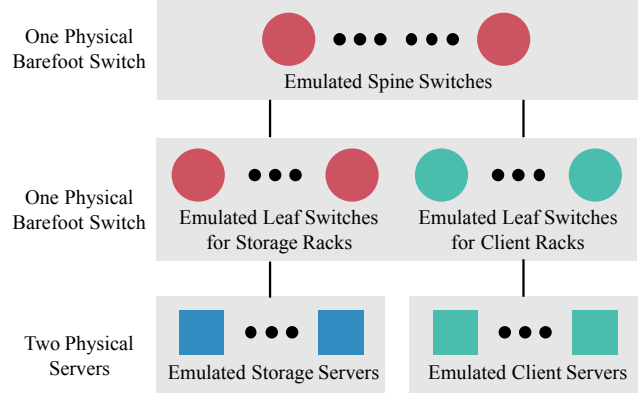

Figure 8: Evaluation setup. The testbed emulates a datacenter with a two-layer leaf-spine network by dividing switches and servers into multiple logical partitions.

Comparison. To demonstrate the benefits of DistCache, we compare the following mechanisms in the experiments: DistCache, CacheReplication, CachePartition, and NoCache. As described in §2.2, CacheReplication is to replicate the hot objects to all the upper layer cache nodes, and CachePartition partitions the hot objects between nodes. In NoCache, we do not cache any objects in both layers. Note that CachePartition performs the same as only using NetCache for each rack (i.e., only caching in the ToR switches).

6.2 Performance for Read-Only Workloads

We first evaluate the system performance of DistCache. By default, we use 32 spine switches and 32 storage racks. Each rack contains 32 servers. We populate each cache switch with 100 hot objects, so that 64 cache switches provide a cache size of 6400 objects. We use read-only workloads in this experiment, and show the impact of write queries in §6.3. We vary workload skew, cache size and system scale, and compare the throughputs of the four mechanisms under different scenarios.

Impact of workload skew. Figure 9(a) shows the throughput of the four mechanisms under different workload skews. Under the uniform workload, the four mechanisms have the same throughput, since the load between the servers is balanced and all the servers achieve their maximum throughputs. However, when the workload is skewed, the throughput of NoCache significantly decreases, because of load imbalance. The more skewed the workload is, the lower throughput NoCache achieves. CachePartition performs better than NoCache, by caching hot objects in the switches. But its throughput is still limited because of load imbalance between cache switches. CacheReplication provides the optimal throughput under read-only workloads as it replicates hot objects in all spine switches. DistCache provides comparable throughput to CacheReplication by using the distributed caching mechanism. And we will show in §6.3 that DistCache performs better than CacheReplication under writes because of low overhead for cache coherence.
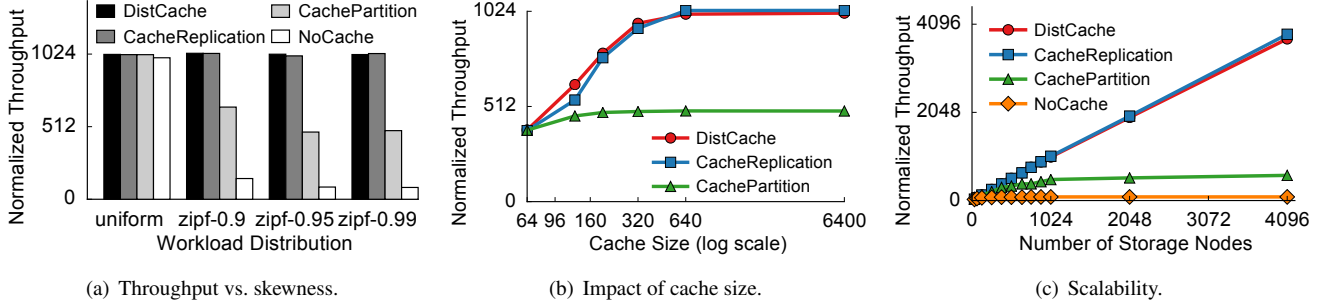
(a) Throughput vs. skewness.   (b) Impact of cache size.   (c) Scalability.

Figure 9: System performance for read-only workloads.

Impact of cache size. Figure 9(b) shows the throughput of the three mechanisms under different cache sizes. CachePartition achieves higher throughput with more objects in the cache. Because the skewed workload still causes load imbalance between cache switches, the benefits of caching is limited for CachePartition. Some spine switches quickly become overloaded after caching some objects. As such, the throughput improvement is small for CachePartition. On the other hand, CacheReplication and DistCache gain big improvements by caching more objects, as they do not have the load imbalance problem between cache switches. The curves of CacheReplication and DistCache become flat after they achieve the saturated throughput.

Scalability. Figure 9(c) shows how the four mechanisms scale with the number of servers. NoCache does not scale because of the load imbalance between servers. Its throughput stops to improve after a few hundred servers, because the overloaded servers become the system bottleneck under the skewed workload. CachePartition performs better than NoCache as it uses the switches to absorb queries to hot objects. However, since the load imbalance still exists between the cache switches, the throughput of CachePartition stops to grow when there are a significant number of racks. CacheReplication provides the optimal solution, since replicating hot objects in all spine switches eliminates the load imbalance problem. DistCache provides the same performance as CacheReplication and scales out linearly.

6.3 Cache Coherence

While read-only workloads provide a good benchmark to show the caching benefit, real-world workloads are usually *read-intensive* [5]. Write queries require the two-phase update protocol to ensure cache coherence, which (*i*) consumes the processing power at storage servers, and (*ii*) reduces the caching benefit as the cache cannot serve queries to hot objects that are frequently being updated. CacheReplication, while providing the optimal throughput under read-only workloads, suffers from write queries, since a write query to a cached object requires the system to update all spine switches. We use the basic setup as the previous experiment, and vary the write ratio.

Since both the workload skew and the cache size would

(a) Throughput vs. write ratio under Zipf-0.9 and cache size 640.

(b) Throughput vs. write ratio under Zipf-0.99 and cache size 6400.
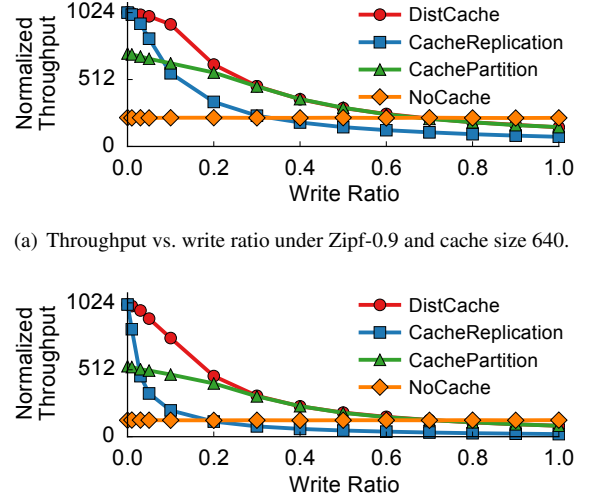
Figure 10: Cache coherence result.

affect the result, we show two representative scenarios. Figure 10(a) shows the scenario for Zipf-0.9 and cache size 640 (i.e., 10 objects in each cache switch). Figure 10(b) shows the scenario for Zipf-0.99 and cache size 6400 (i.e., 100 objects in each cache switch), which is more skewed and caches more objects than the scenario in Figure 10(a). NoCache is not affected by the write ratio, as it does not cache anything (and our rate limiter for the emulated storage servers assumes same overhead for read and write queries, which is usually the case for small values in in-memory key-value stores [33]). The performance of CacheReplication drops very quickly, and it is highly affected by the workload skew and the cache size, as higher skewness and bigger cache size mean more write queries would invoke the two-phase update protocol. Since DistCache only caches an object once in each layer, it has minimal overhead for cache coherence, and its throughput reduces slowly with the write ratio. The throughputs of the three caching mechanisms eventually become smaller than that of NoCache, since the servers spend extra resources on the cache coherence. Thus, in-network caching should be disabled for write-intensive workloads, which is a general guideline for many caching systems.
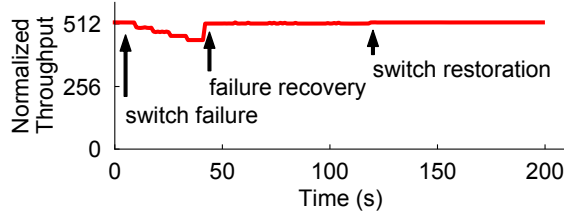
Figure 11: Time series for failure handling.

Switches	Match Entries	Hash Bits	SRAMs	Action Slots
Switch.p4	804	1678	293	503
Spine	149	751	250	98
Leaf (Client)	76	209	91	32
Leaf (Server)	120	721	252	108

Table 1: Hardware resource usage of DistCache.

6.4 Failure Handling

We now evaluate how DistCache handles failures. Figure 11 shows the time serious of this experiment, where x-axis denotes the time and y-axis denotes the system throughput. The system starts with 32 spine switches. We manually fail four spine switches one by one. Since each spine switch provides 1/32 of the total throughput, after we fail four spine switches, the system throughput drops to about 87.5% of its original throughput. Then the controller begins a failure recovery process, by redistributing the partitions of the failed spine switches to other alive spine switches. Since the maximum throughput the system can provide drops to 87.5% due to the four failed switches, the failure recovery would have no impact if all alive spine switches were already saturated. To show the benefit of the failure recovery, we limit the sending rate to half of the maximum throughput. Therefore, after the failure recovery, the throughput can increase to the original one. Finally, we bring the four failed switches back online.

6.5 Hardware Resources

Finally, we measure the resource usage of the switches. The programmable switches we use allow developers to define their own packet formats and design the packet actions by a series of match-action tables. These tables are mapped into different stages in a sequential order, along with dedicated resources (e.g., match entries, hash bits, SRAMs, and action slots) for each stage. DistCache leverages stateful memory to maintain the cached key-value items, and minimizes the resource usage. Table 1 shows the resource usage of the switches with the caching functionality. We show all the three roles, including a spine switch, a leaf switch in a client rack, and a leaf switch in a storage rack. Compared to the baseline Switch.p4, which is a fully functional switch, adding caching only requires a small amount of resources, leaving plenty room for other network functions.

7 Related Work

Distributed storage. Distributed storage systems are widely deployed to power Internet services [1–4]. One trend is to move storage from HDDs and SDDs to DRAMs for high performance [19, 20, 34, 35]. Recent work has explored both hardware solutions [36–47] and software optimizations [33, 48–52]. Most of these techniques focus on the single-node performance and are orthogonal to DistCache, as DistCache focuses on the entire system spanning many clusters.

Load balancing. Achieving load balancing is critical to scale out distributed storage. Basic data replication techniques [25, 53] unnecessarily waste storage capacity under skewed workloads. Selective replication and data migration techniques [54–56], while reducing storage overhead, increase system complexity and performance overhead for query routing and data consistency. EC-Cache [31] leverages erasure coding, but since it requires to split an object into multiple chunks, it is more suitable for large objects in data-intensive applications. Caching is an effective alternative for load balancing [9–11]. DistCache pushes the caching idea further by introducing a distributed caching mechanism to provide load balance for large-scale storage systems.

In-network computing. Emerging programmable network devices enable many new in-network applications. IncBricks [57] uses NPUs as a key-value cache. It does not focus on load balancing. NetPaxos [58, 59] presents a solution to implement Paxos on switches. SpecPaxos [60] and NOPaxos [61] use switches to order messages to improve consensus protocols. Eris [62] moves concurrency control to switches to improve distributed transactions.

8 Conclusion

We present DistCache, a new distributed caching mechanism for large-scale storage systems. DistCache leverages independent hash functions for cache allocation and the power-of-two-choices for query routing, to enable a "one big cache" abstraction. We show that combining these two techniques provides provable load balancing that can be applied to various scenarios. We demonstrate the benefits of DistCache by the design, implementation and evaluation of the use case for emerging switch-based caching.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SOSP*, October 2003.

[2] 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 *ACM SOSP*, October 2007.

[3] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, *et al.*, "Finding a needle in Haystack: Facebook's photo storage.," in *USENIX OSDI*, October 2010.

[4] 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 *USENIX NSDI*, April 2013.

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

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM Symposium on Cloud Computing*, June 2010.

[7] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse, "Characterizing load imbalance in real-world networked caches," in *ACM SIGCOMM HotNets Workshop*, October 2014.

[8] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites," in *WWW*, May 2002.

[9] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect: Provable load balancing for randomly partitioned cluster services," in *ACM Symposium on Cloud Computing*, October 2011.

[10] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with SwitchKV," in *USENIX NSDI*, March 2016.

[11] 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 *ACM SOSP*, October 2017.

[12] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, October 2001.

[13] 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," *CoRR*, vol. abs/1901.08200, 2019.

[14] S. P. Vadhan *et al.*, "Pseudorandomness," *Foundations and Trends® in Theoretical Computer Science*, vol. 7, no. 1–3, pp. 1–336, 2012.

[15] J. Nešetril, "Graph theory and combinatorics," *Lecture Notes, Fields Institute*, pp. 11–12, 2011.

[16] S. Foss and N. Chernova, "On the stability of a partially accessible multi-station queue with state-dependent routing," *Queueing Systems*, 1998.

[17] R. D. Foley and D. R. McDonald, "Join the shortest queue: stability and exact asymptotics," *Annals of Applied Probability*, pp. 569–607, 2001.

[18] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, May 1998.

[19] "Redis data structure store." https://redis.io/.

[20] "Memcached key-value store." https://memcached.org/.

[21] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "CONGA: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM*, August 2014.

[22] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *ACM SOSR*, March 2016.

[23] "Barefoot Tofino." https://www.barefootnetworks.com/technology/#tofino.

[24] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[25] 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 *ACM Symposium on Theory of Computing*, May 1997.

[26] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *ACM SOSP*, October 2001.

[27] 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 CCR*, July 2014.

[28] "Barefoot P4 Studio." https://www.barefootnetworks.com/products/brief-p4-studio/.

[29] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-min Sketch and Its Applications," *J. Algorithms*, 2005.

[30] "Hiredis: Redis library." https://redis.io/.

[31] 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 *USENIX OSDI*, November 2016.

[32] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *ACM SIGMOD*, May 1994.

[33] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *USENIX NSDI*, April 2014.

[34] "Amazon DynamoDB accelerator (DAX)." https://aws.amazon.com/dynamodb/dax/.

[35] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud storage system," *ACM Transactions on Computer Systems*, August 2015.

[36] M. Blott, K. Karras, L. Liu, K. A. Vissers, J. Bär, and Z. István, "Achieving 10Gbps line-rate key-value stores with FPGAs," in *USENIX HotCloud Workshop*, June 2013.

[37] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA Memcached appliance," in *ACM/SIGDA FPGA*, February 2013.

[38] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing SoC accelerators for Memcached," in *ACM/IEEE ISCA*, June 2013.

[39] 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 *ACM SOSP*, October 2017.

[40] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *ACM SIGCOMM*, August 2014.

[41] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *USENIX ATC*, June 2016.

[42] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *USENIX NSDI*, April 2014.

[43] S. Li, K. Lim, P. Faraboschi, J. Chang, P. Ranganathan, and N. P. Jouppi, "System-level integrated server architectures for scale-out datacenters," in *IEEE/ACM MICRO*, December 2011.

[44] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, *et al.*, "Scale-out processors," in *ACM/IEEE ISCA*, June 2012.

[45] A. Gutierrez, M. Cieslak, B. Giridhar, R. G. Dreslinski, L. Ceze, and T. Mudge, "Integrated 3D-stacked server designs for increasing physical density of key-value stores," in *ACM ASPLOS*, March 2014.

[46] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *ACM ASPLOS*, March 2014.

[47] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *ACM/IEEE ISCA*, June 2015.

[48] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A fast array of wimpy nodes," in *ACM SOSP*, October 2009.

[49] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *EuroSys*, April 2014.

[50] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent memcache with dumber caching and smarter hashing," in *USENIX NSDI*, April 2013.

[51] V. Vasudevan, M. Kaminsky, and D. G. Andersen, "Using vector interfaces to deliver millions of IOPS from a networked key-value storage server," in *ACM Symposium on Cloud Computing*, October 2012.

[52] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *ACM SOSP*, October 2011.

[53] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *ACM SOSP*, October 2001.

[54] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *EuroSys*, April 2015.

[55] 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 *CloudDB*, October 2012.

[56] 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," in *VLDB*, November 2014.

[57] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward in-network computation with an in-network cache," in *ACM ASPLOS*, April 2017.

[58] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "NetPaxos: Consensus at network speed," in *ACM SOSR*, June 2015.

[59] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *SIGCOMM CCR*, April 2016.

[60] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing distributed systems using approximate synchrony in data center networks," in *USENIX NSDI*, May 2015.

[61] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say NO to Paxos overhead: Replacing consensus with network ordering," in *USENIX OSDI*, November 2016.

[62] J. Li, E. Michael, and D. R. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *ACM SOSP*, October 2017.