C2DN: How to Harness Erasure Codes at the Edge for Efficient Content Delivery

Juncheng Yang, *Carnegie Mellon University;* Anirudh Sabnis, *University of Massachusetts, Amherst;* Daniel S. Berger, *Microsoft Research and University of Washington;* K. V. Rashmi, *Carnegie Mellon University;* Ramesh K. Sitaraman, *University of Massachusetts, Amherst, and Akamai Technologies*

https://www.usenix.org/conference/nsdi22/presentation/yang-juncheng

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

C2DN: How to Harness Erasure Codes at the Edge for Efficient Content Delivery

Juncheng Yang[1], Anirudh Sabnis[2], Daniel S. Berger[3], K. V. Rashmi[1], Ramesh K. Sitaraman[2,4]

[1]Carnegie Mellon University
[2]University of Massachusetts, Amherst
[3]Microsoft Research and University of Washington
[4]Akamai Technologies

Abstract

Content Delivery Networks (CDNs) deliver much of the world's web and video content to users from thousands of clusters deployed at the "edges" of the Internet. Maintaining consistent performance in this large distributed system is challenging. Through analysis of month-long logs from over 2000 clusters of a large CDN, we study the patterns of server unavailability. For a CDN with no redundancy, each server unavailability causes a sudden loss in performance as the objects previously cached on that server are not accessible, which leads to a miss ratio spike. The state-of-the-art mitigation technique used by large CDNs is to replicate objects across multiple servers within a cluster. We find that although replication reduces miss ratio spikes, spikes remain a performance challenge. We present C2DN, the first CDN design that achieves a lower miss ratio, higher availability, higher resource efficiency, and close-to-perfect write load balancing. The core of our design is to introduce erasure coding into the CDN architecture and use the parity chunks to re-balance the write load across servers. We implement C2DN on top of open-source production software and demonstrate that compared to replication-based CDNs, C2DN obtains 11% lower byte miss ratio, eliminates unavailability-induced miss ratio spikes, and reduces write load imbalance by 99%.

1 Introduction

Content Delivery Networks (CDNs) [20] carry more than 70% of Internet traffic and continue to grow [19]. Large CDNs achieve this by operating thousands of clusters deployed worldwide so that users can download content with low network latency. When a user requests an object, the CDN routes the request to a server proximal to the user [15]. If the server contains the requested object in its cache, the user experiences a fast response (*cache hit*). If no server within the cluster has the object in cache (*cache miss*), the object is fetched from a remote cluster which could be another CDN cluster or the origin (i.e., the content provider).

Detrimental effects of cache misses. Cache misses have three detrimental effects. First, they degrade performance by increasing the *content download times* experienced by the user, as each object incurring a cache miss would have to be downloaded over the WAN from a remote server. Second, cache miss can result in additional traffic between the CDN cluster and the origin, which is a significant bandwidth cost for CDN operators. Third, if more cache misses are served from the origin, content providers need to provision more servers with higher network bandwidth. Consequently, a CDN's goal is to minimize the miss ratio and maintain a low miss ratio over time for all content providers.

Why tail performance matters. The design goal of a CDN is to consistently improve download performance for *all* objects on a content provider's site, in *every* time window, and for *each* client location. The performance improvement is viewed as a "speedup" that the CDN provides over the content provider's origin, i.e., it can be quantified as the ratio of the time to download an object directly from origin (without the CDN) to the time to download the same object from the CDN. A CDN's goal is to provide a significant average speedup in every time window (say, 5-minute window) and at each client location. A spike in the miss ratio in a single cluster could violate these performance goals, *even if that spike is short-lived and impacts only a subset of the objects*. That is because the CDN likely offers no speedup over origin for any client download that is a cache miss, and indeed a short-lived spike in miss ratio could drastically decrease the average speedup provided by the CDN during a 5-minute period.

The challenge of frequent server unavailabilities. Due to stringent performance goals, servers are continuously monitored by the cluster's load balancer. A server is declared to be "unavailable" and (temporarily) taken out of service if it is deemed incapable of serving content to users within specified performance bounds. By analyzing a month long logs from the load balancers in over 2000 clusters of a large CDN, we find that server unavailability is very common in CDN edge clusters. When a server is unavailable, the objects stored in its cache are not available to serve user requests. Unless the requested objects can be retrieved from other servers within the cluster, these objects need to be fetched from a remote server, resulting in a spike in the miss ratio, potentially causing a violation of performance guarantees.

Limitations of the state-of-the-art approaches. To tolerate server unavailabilities, the state-of-the-art approach adopted by large CDNs is to replicate objects across two servers within a cluster. We found that this approach has three significant limitations. First, we find that object replication does not eliminate the miss ratio spike following a server unavailability event. The reason is that the replica of the object (within the cluster) may no longer be present due to eviction from its cache. Second, replicating objects is space-inefficient as the CDN effectively has to provision twice the cache capacity, which is challenging due to the accelerating growth in CDN traffic. Third, we observe a significant *write* imbalance between servers due to DNS based load balancing. This imbalance increases SSD read latency and reduces SSD lifetime [22, 61].

Bringing together efficiency and high availability. In this paper, we present C2DN[1], a CDN design that achieves both high availability and high resource efficiency. To achieve high resource efficiency, we apply erasure coding to large cached objects. This requires overcoming multiple CDN-specific challenges such as eviction of object chunks due to write rate imbalances. In fact, we show that a naive application of erasure coding fails to achieve the goal. The core of our design is a new technique that enables CDNs to balance eviction rates and write loads across servers in each cluster. We exploit the fact that erasure coding enables more flexibility in assigning chunks to multiple servers. Our key insight here is that the chunk assignment can be reduced to a known mathematical optimization problem, called Max Flow Problem.

The core contributions of C2DN are a novel chunk placement scheme for consistent-hashing-based load balancing in CDN clusters and a low-overhead implementation of erasure coding for CDNs that can serve the different traffic requirements of production systems. Specifically, by solving an instance of the Max Flow problem, we assign objects with *near-optimal balance* in eviction and write rates for CDN servers and their SSDs. As a consequence, C2DN can reduce storage overheads and bandwidth costs. Finally, equal write rates across servers essentially function as a cluster-wide distributed wear-leveling for the servers' SSDs, significantly extending lifetimes.

Our contributions. We make the following contributions.

1. We show that server unavailability is common in CDN clusters by analyzing a month-long trace from over 2000 load balancers of a large CDN. We show that the state-of-the-art approach of replicating objects within a cluster does not eliminate miss ratio spikes after a server unavailability events.

2. We design C2DN with a hybrid redundancy scheme using replication and erasure coding, along with a novel approach for parity placement. C2DN reduces the storage overhead of providing fault tolerance, and hence lowers the miss ratio. Moreover, by leveraging the parity assignment, C2DN balances the write loads and eviction rates across cache servers.

3. We implement C2DN on top of the Apache Traffic Server (ATS) [7] and evaluate it using production traces. We show that C2DN provides 11% miss ratio reduction compared to the state-of-the-art, and C2DN eliminates the miss ratio spikes caused by server unavailabilities. Further, C2DN decreases write load imbalance between servers by 99%.

[1]C2DN stands for Coded Content Delivery Network.

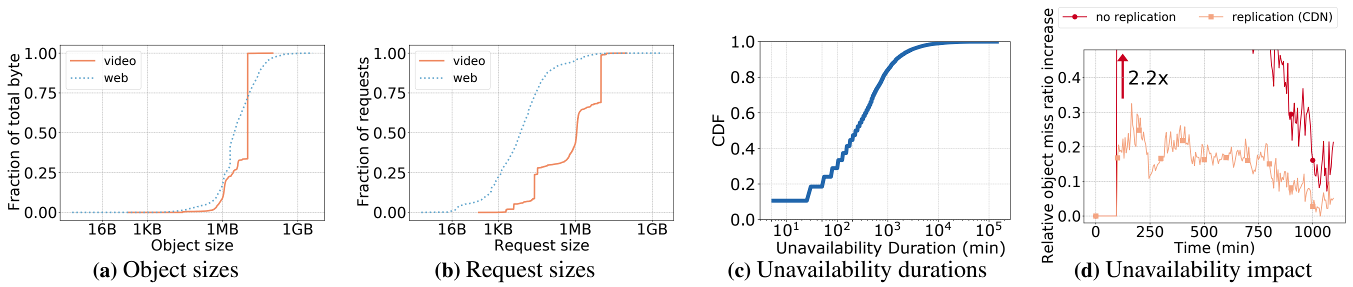
2 Background

We describe CDN architecture, performance, and cost factors.

CDN Architecture. A CDN is a large distributed system with hundreds of thousands of servers deployed around the world [20, 50]. The servers are grouped into *clusters*, where each cluster is deployed within a data center on the edge of the Internet. The CDN cluster caches content and serves it on behalf of *content providers*, such as e-commerce sites, entertainment portals, social networks, news sites, media providers, etc. By caching content in server clusters proximal to the end users, a CDN improves performance by providing faster download times for clients. Unlike storage systems, CDN servers do not store the original content copies. When the requested content is not available in the cluster (cache miss), the content is retrieved from other CDN cluster or the origin servers operated by the content provider.

Bucket-based request routing. When a user requests an object, such as a web page or video, the *global load balancer* of the CDN routes the request to a cluster that is proximal to the user [15]. Next, the *local load balancer* within the cluster routes the request to one or more servers within the chosen cluster that can serve the requested object. As an example, in Akamai's CDN, these routing steps are performed as DNS lookups. A content provider CNAMEs its domain name (e.g., for all of its media objects) to a sub-domain whose authoritative DNS server is the CDN's global load balancer. At the global load balancer, this sub-domain is CNAME'd to a cluster-local load balancer that assigns the sub-domain to a cluster server using consistent hashing [43].

CDN request routing stands in contrast to sharding in key-value caches, such as Memcached and Redis, where consistent hashing is often applied at a per-object granularity [49, 81–83]. *In CDNs, load balancing decisions are taken on the granularity of groups of objects called buckets*. Each bucket, in a DNS-based load balancer, correspond to a domain name that is resolved to obtain one or more server IPs that host objects in that bucket. This resolution is computed using consistent hashing. Since the number of buckets is limited in the range of 100s, the computation is performed and cached when a cluster server becomes available or unavailable.



(a) Object sizes **(b)** Request sizes **(c)** Unavailability durations **(d)** Unavailability impact

Figure 1: a) Size distributions show that large objects contribute to most of the unique bytes and b) most requests are for small objects. c) Server unavailabilities are mostly transient. d) Object miss ratios spike after server unavailability both with and without the state-of-the-art replication.

CDN Performance Requirements. A CDN aims to serve content faster than a customer's origin by a specified speedup factor. This factor is commonly part of a service level agreement (SLA) between the CDN and the content provider. The SLA is monitored by recording download times from a globally distributed set of locations for the same content using both CDN and origin servers. Hence, the goal is to ensure good "tail" performance in *every* time interval for *every* content provider from *every* cluster.

Operating Costs of a CDN. CDNs seek to minimize the operating cost, which consists of the following main categories. (i) *Bandwidth* : A major component of the operating cost of a CDN is bandwidth, accounting for roughly 25% of operating costs. The bandwidth cost can be further broken down, the bandwidth cost caused by cache miss traffic called *midgress* [69] that accounts for roughly 20%, while the rest is the cost of *egress* i.e., the traffic from the CDN servers to clients. CDNs have a great cost incentive to reduce the byte miss ratio and the midgress traffic since a CDN gets paid by content providers for the traffic to end users. The midgress traffic between CDN clusters and the origin is purely a cost overhead for the CDN. Even modest reductions in midgress translate into large cost savings since the bandwidth costs tens of millions of dollars per year for a large CDN [69].
(ii) SSD wearout: A second major cost component is server deprecation which accounts for about 25% of the operating cost of a large CDN. Hardware replacements are particularly expensive for small edge clusters due to the large geographic footprints of CDNs. SSDs are a key component due to the high IOPS requirements of CDN caching. Unfortunately, using SSDs in caching applications is challenging due to their limited write endurance [9, 22, 42, 67, 70]. With deployments of TLC and QLC SSDs, reducing SSD write rates has become even more critical. Besides reducing the average write rate within a cluster, CDNs also seek to reduce the variance of write rates of different servers and their SSDs. Large variance leads to some SSDs not achieving their intended lifetime (e.g., 3 years) as well as high tail latency (see §3.4). Consequently, CDNs seek to reduce the peak write rate, ideally balancing write rates across all SSDs in a cluster.

Per server load (TB)	Max	Min	Mean	Max/min
Weekly read	225.2	167.9	191.2	1.3
Weekly write	16.54	6.69	12.57	2.5

Table 1: Read and write load for a 10-server production cluster.

3 Production CDN Trace Analysis

This section motivates the design of C2DN by analyzing three sets of traces from production Akamai clusters.

We collected request traces from two typical Akamai 10-server-clusters (cluster cache size 40 TB), one mainly serving web traffic and the other mainly serving video traffic. These traces comprise anonymized loglines for every request from every server over a period of 7 and 18 days, respectively. The **web trace** totals 6 billion requests (1.7 PB) for 273 million unique objects (79.8 TB). The **video trace** totals 600 million requests (2.1 PB) for 130 million unique objects (224 TB).

Additionally, we collected availability traces from 2190 Akamai clusters over 31 days. The trace consists of snapshots taken every 5 minutes from each cluster's local load balancer. Each snapshot contains the number of available servers as determined by the load balancer. The smallest cluster has two servers, the largest cluster has over 500 servers, and the median cluster size is 17 servers. We observe that cluster size has a wide range, and around 40% of clusters have fewer than or equal to 10 servers. We plot the distribution of cluster size in Fig. 10 in Appendix 10.1.

3.1 Diversity in workloads and object sizes

CDNs mix different types of traffic in clusters in order to fully use their resources. For example, different "classes" of traffic with small and large object sizes, such as web assets and video-on-demand, are mixed to balance the utilization of the cluster's CPUs as well as network and disk bandwidth [68]. Consequently, object sizes vary widely [10]. Figures 1a and 1b show the size distribution for our production traces, weighted by unique objects and by request count, respectively. As expected, object sizes vary from a few bytes to a few GBs. Fig 1a shows that *the majority of traffic and cache space is used by large objects*. Furthermore, objects smaller than 1 MB make up less than 15% and 12% of the total working set in web and video, respectively. Fig 1b shows that *the majority of the requests are for small objects* with 95% of requests in web-dominant workload smaller than 1 MB, and 50% of requests in video-dominant workload smaller than 1 MB.

3.2 Unavailability is common and transient

Unavailability is common. Across all clusters, server unavailabilities occur in 45.2% of the 5-minute snapshots. For clusters with only ten servers (same size as the cluster we collect request traces from), we observe that 30.5% of 5-min time snapshots show server unavailability. Moreover, we observe that unavailability affects only a small number of servers at any given time: 85% of unavailabilities affect less than 10% of servers in large clusters, and 84% of unavailabilities affect no more than a single server in a ten-server cluster.

These unavailability rates can appear high compared to published failure rates in large data centers [25, 46, 54, 56, 59] and HPC-systems [65]. However, environmental conditions can be more challenging in small edge clusters. For example, edge locations often have less efficient cooling systems than highly optimized hyperscale data centers; edge clusters also have less power redundancy, such as redundant battery and generator backups [50]. Moreover, CDN clusters employ a rigorous definition of server unavailability. When a server does not meet the performance requirement, it is deemed as unavailable by the load balancer. These types of unavailability are rarely reported by data centers and HPC systems. Unfortunately, the unavailability logs do not provide a causal breakdown of failure events.

Unavailability is mostly transient. Fig. 1c shows a CDF of the durations of unavailabilities. We observe that unavailabilities can last between 20 minutes and 24 hours with a median duration of 200 minutes. These short unavailabilities are mostly caused by performance degradation, such as unexpected server overload and software issues (e.g., application/kernel bugs or upgrades). Besides, we observe a long tail of unavailability durations, with around 16% exceeding 24 hours and 2% exceeding an entire week. These cases may be related to hardware issues. Qualitatively, our observations are similar to storage systems in the sense that unavailabilities are common and most unavailabilities are not permanent.

3.3 Mitigating unavailability is challenging

Upon detecting an unavailability, the load balancer removes the corresponding server from the consistent hash ring and reassigns their buckets to other servers [43]. We evaluate how a bucket's object miss ratio is affected by unavailability using the video trace. Fig. 1d shows that the object miss ratio in a CDN cluster *without any redundancy* increases by more than 2× relative to no unavailability over the same time period. *This spike disproportionally affects a small group of content providers because of bucket-based routing* (§3.1). The high latency resulting from cache misses can lead to SLA violations.

The state-of-the-art mitigation technique for server unavailability at large CDNs is *replicating* buckets across two servers[2]. When one server becomes unavailable, requests are routed to the other server, likely to hold the object. Fig. 1d shows that replication reduces the intensity of the miss ratio spike. However, we find that replication does not remove the miss ratio spike. *In contrast to storage systems, where replication guarantees durability, in CDN clusters, servers perform cache evictions independently.* Objects that are admitted to two caches at the same time may be evicted at different times. This is particularly common if the two caches evict objects at very different rates, making replication ineffective. We next discuss why this case is more common than one might expect.

3.4 The need for write load balancing

We measure the read and write load balance across servers in a CDN cluster. To make the analysis independent from eviction decisions, we present the read and write rates based on compulsory misses from the web trace [3]. Table. 1 shows that the server with the highest read load serves 1.3× more traffic than the server with the lowest read load. The server with the highest write load writes around 2.5× more bytes than the server with the lowest write load.

Write load imbalance causes three problems. First, imbalance reduces the effectiveness of replication. A server with a 2.5× higher write rate also has a 2.5× higher eviction rate. So, a newly admitted object will traverse the cache with the highest write load 2.5× faster than the one with the least write load. Consequently, buckets mapped to these servers will have many objects for which only a single copy is cached in the cluster. We find that for 25% of objects, only a single copy exists in the cluster, which leads to the miss ratio spike observed during unavailabilities (Fig. 1d). Second, SSD write load imbalance often causes high tail latency. Specifically, high write rates frequently trigger garbage collection, which can delay subsequent reads by tens of milliseconds [11, 77, 78, 80]. These delays are significant enough to have been recognized as a problem by multiple CDN operators [61]. Third, the imbalance can lead to short SSD lifetimes due to concentrated writes on some SSDs, and thus higher replacement rates [9, 22], which increases CDN cost (§2).

4 C2DN System Design

C2DN's design goals are to: (1) eliminate miss ratio spikes

[2]For operational flexibility, CDNs do not replicate servers as primary/backup. CDNs implement replication using additional virtual nodes for a bucket on the consistent hash ring [36, 47].

[3]Compulsory misses are cache admissions forced by objects not previously seen in the trace (underestimating the real miss and write rate). However, more compulsory misses only lead to more writes and evictions. Therefore, write rates are often proportional to compulsory misses.

caused by server unavailability, and (2) balance write loads across servers in the cluster. Erasure coding is a promising tool to improve availability under server unavailability. We first describe a naive implementation, called C2DN-NoRebal, based on a straightforward application of erasure coding (§4.1). C2DN-NoRebal fails to achieve the targeted goals, and we identify write and eviction imbalance as the key challenge. We then describe a new technique to overcome this challenge (§4.2) that exploits the unique aspects of the use of erasure coding in the context of CDNs.

4.1 Erasure coding and C2DN-NoRebal

Erasure coding is widely used in production storage systems for providing *high availability* with *low resource overhead* [32, 35, 48, 48, 54, 56]. Conceptually, erasure coding an object involves dividing the object into K *data chunks* and creating P parity chunks, which are mathematical functions of the data chunks. Such a scheme, called a (K,P) coding scheme, enables the system to decode the full object from any K out of the $K+P$ chunks. Thus, caching $K+P$ chunks on different servers provides tolerance to P server unavailabilities. As individual chunks are only a fraction $1/K$ of the original object's size, coding reduces space overhead compared to replicating full objects[4].

As CDNs use bucket-based routing (§2), coding needs to be applied at the level of buckets rather than objects. Specifically, the K data chunks of all the objects belonging to a bucket are grouped into K distinct *data buckets* respectively. Similarly, the corresponding P parity chunks are grouped into P distinct *parity buckets*. These buckets (data and parity) are each assigned to a distinct server in the cluster. Note that while the routing happens at the level of buckets, requests are still served at the level of objects. Hence we will use the term *buckets* in the context of assignment and *chunks* in the context of serving specific objects.

The application of erasure coding to CDNs is shown in Fig. 2a. To serve a user request, a server reads one chunk from the local cache and at least $K-1$ chunks from other servers to reconstruct the requested object. To find the location of data and parity chunks, *C2DN-NoRebal* relies on a simple extension of bucket-based consistent hashing. The location of the first chunk is the server the bucket containing the object hashes to. Then, subsequent $K+P-1$ chunks are read from the subsequent $K+P-1$ servers on the consistent hash ring.

Owing to the reduced storage overhead, C2DN-NoRebal provides cost benefits by reducing the average byte miss ratio when compared to replication (as seen in our experiments in §6). However, C2DN-NoRebal *fails to eliminate the object miss ratio spike during unavailability* (§6). Specifically, we find that coded caches are even more sensitive to write load imbalance than replication. For replication, eviction rate imbalance may cause the second (backup) copy to be evicted, which is required when a server becomes unavailable. Whereas for a coded cache, eviction rate imbalance could lead to any of the individual chunks being evicted, which leads to an effect we call *partial hits*: less than K chunks of the object are cached in the cluster, and this prohibits the reconstruction of the object. A partial hit only requires fetching the missing chunks, but incurs the same round-trip-time latency as a miss and thus does not provide a speedup. Further, partial hits become even more frequent during server unavailability, thus deeming C2DN-NoRebal less effective.

[4]The space overhead of an (K,P) coding scheme is $\frac{K+P}{K}$. For example, for $K=3$, $P=1$, space overhead is $1.33\times$ as opposed to $2\times$ in two-replication.

4.2 Parity rebalance and C2DN

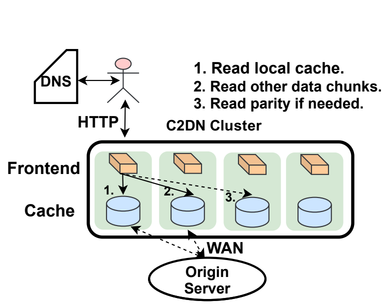
Having identified write imbalance as a key challenge for erasure coding in CDNs, we next show how we exploit parities in overcoming these imbalances. Our main idea is to assign *parity buckets* to servers in a way that mitigates the write load imbalance caused by *data bucket* assignment.

Like the state-of-the-art in CDNs and C2DN-NoRebal, C2DN applies consistent hashing to assign the data buckets (Fig. 2b). We define a server's data write load as the number of bytes written (i.e., admitted) to cache, counting only data buckets. We also define a bucket's parity write load as the bytes written counting only parity buckets. Every server records data write load and each bucket's parity write load since the cluster's last unavailability event. After an unavailability event, parity buckets are reassigned by the load balancer using this information. The load balancer calculates an assignment of parity buckets to servers to balance write load. This assignment is a non-trivial calculation as not every assignment is feasible: parity chunks cannot be assigned to a server that holds a data chunk of the same object. In general, C2DN's parity bucket assignment problem is NP-hard by reduction from the Generalized Assignment Problem [14].

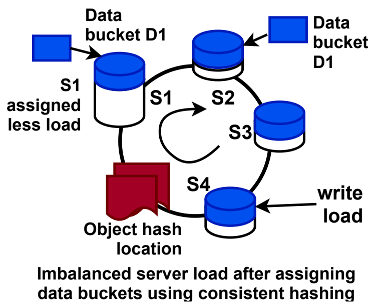
C2DN's parity bucket assignment algorithm. We obtain an approximate solution in polynomial time using a MaxFlow formulation (Fig. 2c). The solution provides us with feasible server assignments for each parity bucket. We empirically observe that by assigning the parity bucket to the least loaded server among the feasible servers, the write load on each server is well balanced. The inputs to the algorithm are:

1. parity write load of bucket n (s_n),
2. data write load on server i (l_i),
3. total write load on the cluster (W),
4. current assignment of data buckets to servers,
5. available servers in the cluster ($\mathcal{A}$).

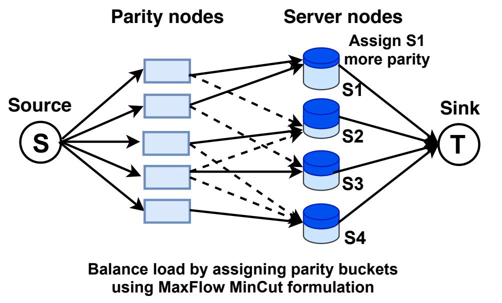
The flow graph (Fig. 2c) is constructed using a source-node (S), parity-nodes corresponding to each parity bucket, server-nodes corresponding to each server in the cluster, and a sink-node (T). We add an edge from the source-node (S) to each parity-node n with a capacity equal to the bucket's parity write load (s_n). We add edges from parity-nodes to the server-nodes if the corresponding parity bucket can be placed on that



(a)



(b)


Parity nodes
Server nodes
Assign S1 more parity
Source
S
S1
S2
S3
S4
Sink
T
Balance load by assigning parity buckets using MaxFlow MinCut formulation

(c)

Figure 2: a) **Architecture of C2DN**; b) **C2DN bucket assignment** C2DN assigns data buckets using consistent hashing, which guarantees a consistent mapping across unavailabilities, but causes load imbalance; c) **Parity rebalance.** Write load imbalance is mitigated by assigning parity buckets to balance the load using a MaxFlow formulation.

server, i.e., the data chunks of the bucket are not assigned to the server. The capacity of these edges is again the bucket's parity write load (s_n). Finally, we add edges from server-nodes to the sink-node (T) with a capacity equal to the server's remaining write load budget, which is $\max(\left\lceil \frac{W}{|\mathcal{A}|} \right\rceil - l_i, 0)$.

After solving MaxFlow(S,T), C2DN iterates over parity buckets. Each parity bucket is assigned to the least loaded server with a positive flow from the parity-node to the server-nodes. This leads to a well-balanced assignment. The assignment is also feasible as no positive flow exists between a parity bucket and the servers holding this bucket's data chunks.

The parity rebalance algorithm is described in more detail via a pseudo-code in Appendix 10.3.

Extension to heterogeneous servers. We incorporate heterogeneous servers by setting the capacity of the edge in the graph between server-nodes to sink-node (T) proportional to the size of the server.

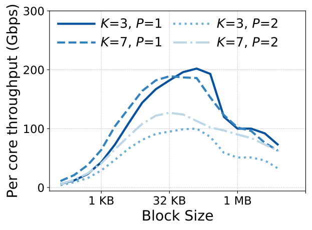
4.3 C2DN resolves partial hits

Having shown how to balance write loads across servers within the cluster, we show that this is sufficient to solve the issue of partial hits. Specifically, we find that the probability of a partial hit diminishes for large caches.

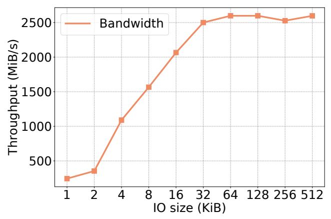
We formulate our proof under the simplifying assumptions of the independent reference model (IRM[5]), which is used widely in caching analysis [5, 10, 23]. While our proof can be extended to a range of eviction policies [44], we assume the Least-Recently-Used (LRU) policy for simplicity. We empirically observe that FIFO, which is used in open-source caches such as Apache Trafficserver [7] and our empirical evaluation in §6, behaves similarly to LRU.

We remark that we *do not require explicit coordination of individual eviction decisions among the caches*. Our theorem states that under IRM, in C2DN, if one chunk of an object is present in a cache, then the other chunks are almost surely present in the other caches.

[5]In the IRM, an object i's requests arrive according to a Poisson process with a rate λ_i, independent of the other objects' requests. With recent theoretical advances [34], our proof can be extended to not assume the IRM.



(a) Decoding throughput



(b) SSD throughput

Figure 3: Microbenchmarks. a) With vector instruction in modern CPU, decoding is very efficient with high throughput, the sub-chunk size to achieve maximum throughput across configurations is around 32-64 KB. b) Modern SSD achieves maximum throughput with I/O size larger than 32 KB.

Theorem 1. *Under IRM and LRU, in C2DN, for an object with chunks $x_1, \ldots, x_n$, for any $1 \le i, j \le n$., and as the cache size grows large*

$$P[\textit{chunk } x_i \textit{ is in cache} \mid \textit{chunk } x_j \textit{ is in cache}\,] \to 1 \qquad (1)$$

Our proof uses the fact that balanced write loads lead to equal *characteristic times* [10, 23, 26, 60], which is the time it takes for a newly requested chunk to get evicted from each server's LRU list. Since data and parity chunks of an object are requested simultaneously and the characteristic time is the same, the chunks are also evicted simultaneously, and partial hits become rare. Details can be found in Appendix 10.2.

5 C2DN Implementation

In addition to design goals (1) and (2), C2DN's implementation seeks to (3) minimize storage/ latency/ CPU overheads and (4) remain compatible with existing systems to facilitate deployment. This entails subtle implementation challenges.

Enabling transparent coding. A key architectural question is which system component encodes and decodes objects into/from data and parity chunks. A natural choice might be to encode objects at origin servers. However, this would require changes to thousands of heterogeneous origin software stacks — a barrier to deployment. Additionally, encoding at the origin would increase origin traffic as each cache miss needs to fetch both data and parity chunks, e.g., with $K = 3$, $P = 1$ the origin traffic would increase by 33%. Thus, C2DN fetches uncoded objects from origins and encodes chunks within the CDN cluster. Additionally, any decoding operation is also performed within the cluster for transparency on the client side.

Selective erasure coding. While encoding and decoding are fast due to broad CPU support for vector operations, the overhead of fetching becomes significant for small objects. As the majority of requests are for small objects (§3.1), we can reduce processing overheads by using replication for small objects. C2DN applies coding to large objects, which account for most of the production cluster's cache space (§3.1). Of course, with selective coding, we now need to count uncoded objects as part of the data write load in §4.2.

To decide the size threshold of coding, we perform two microbenchmarks studying how coding block size affects coding throughput and SSD bandwidth. Fig. 3a shows that even on a five-year-old Skylake Xeon, decoding is very efficient with per-core throughput over 200 Gbps (data fits in CPU cache) at a block size of 32 KB. This benchmark result suggests that decoding will not be a bottleneck at a reasonable block size (e.g., 32 KB) compared to NIC bandwidth. Fig. 3b shows the relationship between SSD bandwidth and I/O size (setup as in §6). We again find that a block size of 32-64 KB achieves the peak SSD bandwidth. Based on these results, C2DN codes object larger than 128KB so that each chunk is at least 42KB for a (3, 1) coding scheme.

This hybrid approach enables load balancing and space efficiency with no overhead for most requests. One might ask why C2DN relies on replication for small objects after §2 showed that replication continues to suffer from miss ratio spikes. We find that erasure coding large objects is sufficient to balance eviction rates (using C2DN's parity rebalance), making replication effective for small objects.

Parity rebalance and parity look up. As described in § 4.2, C2DN formulates the parity bucket assignment problem as a Max Flow problem. We solve the problem using Google-OR [53], which implements the push-relabel algorithm [18]. The time complexity of this algorithm is $O(n_{node}^2 * \sqrt{n_{edge}})$. where n_{node} is the number of nodes (#buckets + #servers) and n_{edge} is the number of edges (≈#buckets × #servers). In production systems, #buckets is in the range of 100s for a 10-server cluster. Thus, the time complexity simplifies to $O(\#buckets^3)$. Empirically, we observe low run times as well, for e.g., for 100 buckets and 10 servers, C2DN's parity bucket assignment runs within 50 μs. Also, note that the parity bucket mapping is calculated in the background (off the critical path) and only when there is an unavailability event. From our analysis, we observe around 5.6 unavailability events on an average day.

Support for large file serving, HTTP streaming, and byte-range requests. To minimize latency, CDNs stream large objects to clients. We achieve compatibility with streaming by subdividing data and parity chunks (for very large objects) into smaller parts which we call sub-chunks. C2DN's encoding and decoding work on the sub-chunk level as shown in Fig. 4. We implement streaming by serving sub-chunks as they become available. For byte-range requests, C2DN fetches the sub-chunks overlapping with the requested byte-range.

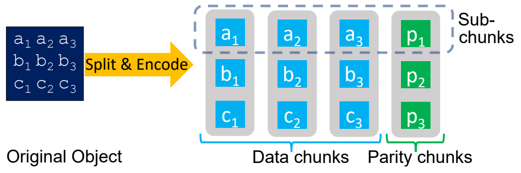


Figure 4: Support for HTTP streaming. C2DN efficiently supports HTTP streaming and byte-range requests by splitting large files into sub-chunks and performs coding on sub-chunks level.

System	Replication(CDN)	C2DN	C2DN reduction
Object miss ratio	0.242	0.227	6.4%
Byte miss ratio	0.118	0.105	11%

Table 2: Object and byte miss ratio from prototype

Delayed fetch of parity sub-chunks. C2DN can serve a request with any K sub-chunks (out of $K+P$). Because serving with data sub-chunks requires no decoding, C2DN first fetches all K data sub-chunks. C2DN only fetches parity sub-chunks after a heuristic wait period to overcome stragglers. We record the time until the first data sub-chunk is returned. If, after an additional 20% wait time, fewer than K data sub-chunks have arrived, C2DN fetches parity sub-chunks.

Hot object cache (HOC). To facilitate serving hot objects, C2DN caches decoded sub-chunks in DRAM so that if an object is popular, it will be served directly and efficiently from DRAM, thus avoiding fetching and possible decoding.

Metadata lookups. In the case of a HOC miss, C2DN needs to know if the object was encoded or replicated. Storage systems can rely on external metadata for this case, which is not available in CDNs. Thus, C2DN stores metadata with each cached object, indicating whether the object is coded or not. On a HOC miss, C2DN first looks up the object in its *local SSD cache*. If the metadata indicates a coded object, C2DN fetches chunks from other caching servers within the cluster. In the case of a local cache miss, C2DN retrieves the object from other CDN clusters or the origin servers, then C2DN serves the object to the end-user, stores it locally, and encodes or replicates based on the object size.

6 Evaluation

We build C2DN on top of Apache Trafficserver and evaluate it via a series of experiments on Amazon EC2. To study a more comprehensive parameter range, we use simulations. The source code of our prototype and simulator is released at `https://github.com/Thesys-lab/C2DN`.

The highlights of our evaluation are: (1) C2DN eliminates miss ratio spikes after unavailabilities. Additionally, C2DN reduces byte miss ratio by 11%, enabling significant bandwidth cost savings at scale. (2) C2DN reduces write load imbalance by 99%. (3) C2DN achieves the same latency, lower average SSD write rates with only a 14% increase in CPU utilization.

6.1 Experimental methodology and setup

Traces. We evaluate C2DN using the two production traces described in §3. In the following sections, we focus on the video trace and present results for the web trace in §6.7.

Prototype evaluation setup. We emulate a CDN's geographic distribution by placing sets of clients, a 10-server CDN cluster, and an origin data center in different AWS regions. CDN servers use i3en.6xlarge VMs with 80 GB in-memory cache and 10 TB disk cache. To reduce WAN monetary bandwidth costs of the experiments, we measure latency via spatial sampling [72, 73] for 2% of requests. The remaining requests are generated in the same region.

Unless specified otherwise, we use Reed-Solomon codes ($K = 3, P = 1$). We only code objects larger than 128 KB (§4). The prototype experiments use four days of requests to warm up caches. Measurements are then taken for three days of requests. This corresponds to replaying 1.18 PB of traffic in total from local and remote clients in each prototype experiment.

Simulation setup. We implement a request-level cluster simulator. While the simulator does not capture system overheads, it is useful in comparing various schemes for the full duration of the trace and for various cache sizes (which are prohibitively expensive to perform using prototype experiments.) Simulations use 18-day long traces (compared to 7 days with the prototype). Unless otherwise stated, the simulator uses the same configuration as the prototype.

Baselines. We compare C2DN to three baselines. **(1) No-replication** does not provide fault tolerance and incurs no space overhead. **(2) Replication (CDN)** replicates each object with two replicas. We use the (CDN) suffix as this is most similar to the approach deployed today. **(3) C2DN-NoRebal** a C2DN variant based on consistent hashing without parity rebalance. In addition to C2DN, which uses one parity chunk and tolerates one unavailability, we have also evaluated C2DN-n5k3 and C2DN-n6k3, which uses two and three parity chunks, and can tolerate two and three unavailabilities, respectively.

6.2 Miss ratio without unavailability

We evaluate miss ratios of the competing systems under normal operation, i.e., *without unavailability*. Table. 2 shows the object miss ratio and byte miss ratio of Replication (CDN) and C2DN obtained from the prototype experiments. We observe that C2DN reduces object miss ratio by 6.4% and byte miss ratio by 11.0%. These improvements are direct results of the reduced storage overhead in C2DN. At a large scale, these improvements lead to significant bandwidth savings.

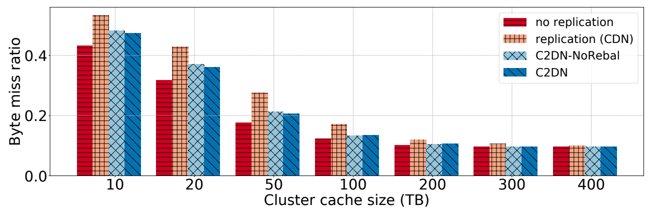


Figure 5: Byte miss ratio of the four systems.

To understand the sensitivity of byte miss ratio improvements to cache size, we show simulation results in Fig. 5. For smaller cache sizes, C2DN improves byte miss ratios by up to 20%. Benefits diminish for cache sizes above 200 TB (5× production cache size). For object miss ratios, the effect is qualitatively similar (Fig. 12 in the appendix). Overall, the reduction in miss ratio bridges the efficiency gap between No-replication and Replication (CDN) and reduces the overhead of providing redundancy in CDN edge clusters.

We also observe that C2DN improves miss ratios compared to C2DN-NoRebal because C2DN balances the write loads (eviction rates) across servers and reduces the probability of partial hits. However, this effect is small, suggesting that most of C2DN's miss ratio reduction comes from reduced storage overhead. The advantage of C2DN over C2DN-NoRebal will become clear in the following section, where we find that C2DN-NoRebal does not provide effective fault tolerance.

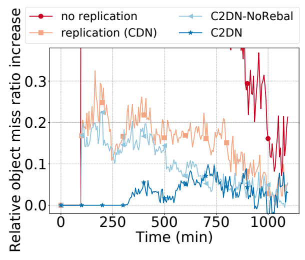
6.3 Miss ratio under unavailability

We now consider unavailabilities and evaluate the object miss ratio as the primary performance metric affecting latency and speedup. A first experiment introduces single unavailability after warming up the cache. We then measure the relative object miss ratio change: $\frac{mr(un)-mr(av)}{mr(av)}$ for each 5 minute time interval, where $mr(un)$ and $mr(av)$ stand for miss ratio with unavailability and without unavailability, respectively. A second experiment considers two simultaneous unavailabilities.

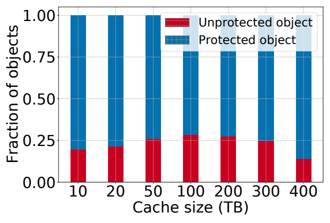
Fig. 6a show the relative object miss ratio increase where the single unavailability event occurs 100 minutes after warmup. As expected, No-replication does not provide fault tolerance, leading to a large (2.2× as seen in 1d) miss ratio spike. Replication (CDN) and C2DN-NoRebal have similar performance with 25% miss ratio spikes.

The miss ratio of C2DN is not affected for several hours after the unavailability event. This is because C2DN with one parity chunk can tolerate one unavailability effectively. In the long term, miss ratios for all systems increase as the cluster's total capacity is reduced. For C2DN, the increase in the miss ratio becomes visible only after around 300 minutes past unavailability. During unavailability, data that should be written to the unavailable servers are written to the other available servers. The extra writes take a long time to impact the miss ratio of clusters with a large cache size. We remark that the exact length of such no performance degradation is not fixed and is dependent on the trace.

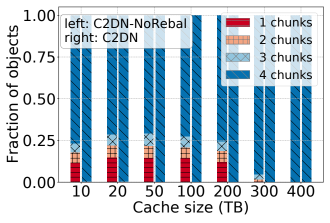
The reason for the miss ratio spike in Replication (CDN)



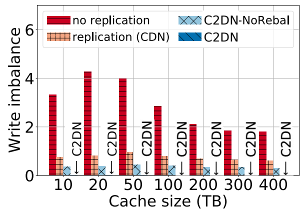
(a)



(b)

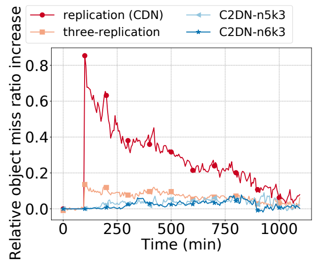


(c)

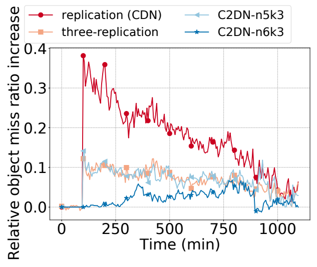


(d)

Figure 6: a) Replicated CDN mitigates unavailability, but still has a spike after unavailability. C2DN mitigates the unavailability spike. b) Servers in the Replicated CDN evict objects independently, and due to write imbalance this leads to unprotected objects. c) Naive coding has similar problems as replication; C2DN solves this problem by parity rebalance. d) Write load imbalance for different systems across various cache sizes.



(a) Affected server 1



(b) Affected server 2

Figure 7: With two simultaneous unavailabilities, two replication (CDN) shows a big spike when the unavailability happens. Three replication and C2DN-n5k3 still show a small spike due to evicted replica/chunk. C2DN-n6k3 completely eliminates the spike.

and C2DN-NoRebal— despite using redundancy — is the severe write and eviction rate imbalance in these systems (§3.3 and 3.4). This imbalance leads to *unprotected* objects: an object is unprotected if only a single copy is cached in the cluster. For C2DN-NoRebal, unprotected objects are objects with fewer than $K+1$ chunks cached in the cluster.

Fig. 6b shows the fraction of (un)protected objects in Replication (CDN). We observe that more than 25% of objects can be unprotected. The fraction of unprotected objects initially increases with cache size and then decreases. This pattern is because only highly popular objects are cached when the cache size is small, and hence the chance of having both replicas is higher. On the other hand, replicas are less likely to be evicted when the cache size is very large. Fig. 6c shows the fraction of unprotected objects in C2DN-NoRebal and C2DN. Since K =3 and P =1, objects with fewer than 4 chunks are unprotected. For caches smaller than 300 TB, up to 24% of the objects in C2DN-NoRebal are unprotected. In contrast, C2DN protects nearly 100% of cached objects across all cache sizes and effectively eliminates miss ratio spikes.

So far, we have only focused on one unavailability. When a CDN operator seeks to tolerate more than one unavailability, C2DN's advantage over replication increases as the space requirements for erasure coding scale significantly better. As an empirical data point, we consider two unavailabilities and compare two-replication and three-replication with C2DN-n5k3 and C2DN-n6k3. C2DN-n5k3 (C2DN-n6k3) uses two

System/server load	Max	Min	Mean	Max/min
CDN write (TB)	16.83	9.26	13.48	1.82
C2DN write (TB)	8.44	8.40	8.42	1.00

Table 3: Write load on servers in Replication (CDN) and C2DN.

(three) parity chunks with 66% (100%) storage overhead and can tolerate two (three) unavailabilities. In contrast, two-replication and three-replication tolerate one and two unavailabilities with 100% and 200% storage overhead, respectively.

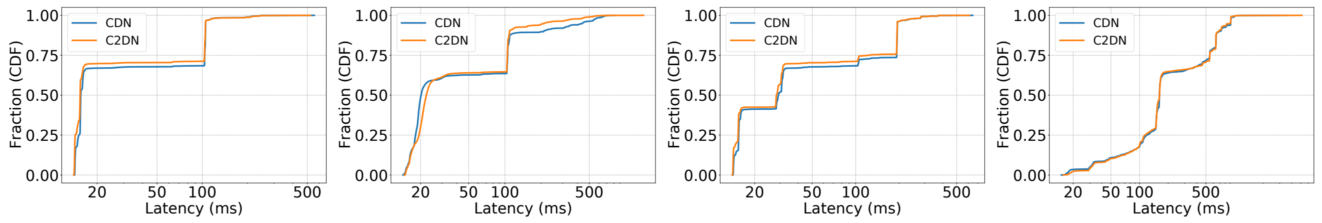
Fig. 7 shows that compared to two-replication, C2DN-n5k3 and three-replication significantly reduce the miss ratio spike from over 80% to less than 20%. Furthermore, the miss ratio spike disappears entirely with C2DN-n6k3, which has the same storage overhead as two-replication.

6.4 Write (Read) load balancing

We quantify how well systems balance write load across servers. Balancing writes is the key to mitigating miss ratio spikes and helps control SSD tail latency and endurance.

Table. 3 shows bytes written per server in our prototype experiments. The busiest server in Replication (CDN) writes 16.8 TB compared to 8.4 TB for the busiest server in C2DN. With half the write rate, C2DN may double SSD lifetime and reduce tail latency by up to an order of magnitude [80]. The write imbalance in Replication (CDN) between peak and minimum write rate is 1.82×. In contrast, the write imbalance in C2DN is less than 1.005×. We also observe that C2DN reduces read imbalance from 1.69× for Replication (CDN) to 1.34×. The read imbalance in C2DN remains as parity rebalancing (§4.2) focuses exclusively on write rate.

We further explore the effects of load balancing across various cache sizes using simulations. If M is the write (read) load on the server with maximum write (read) load and m is the minimum write (read) load across the servers, then write (read) load imbalance $= \frac{M-m}{m}$. Fig. 6d shows that C2DN eliminates write imbalance for all cache sizes. When averaged across cache sizes, C2DN reduces the **write** load imbalance by 99.9% compared to No-replication, 99.8% compared to Replication (CDN), and 99.5% compared to C2DN-NoRebal. C2DN also reduces the **read** load imbalance: by



(a) First byte small objects **(b)** First byte large objects **(c)** Full response small objects **(d)** Full response large objects

Figure 8: First-byte and full response latency of serving small and large objects in CDN and C2DN.

93.9% compared to No-replication, 78.9% compared to Replication (CDN), and 70.5% compared to C2DN-NoRebal on an average across the different cache sizes.

6.5 Latency

We quantify potential latency overheads by measuring the time-to-first-byte (TTFB) and content download time (CDT) of our prototype implementations of C2DN and Replication (CDN). In each case, we separately measure the latency distribution for objects below the 128KB coding threshold ("small" objects) and for objects above the threshold ("large" objects). Fig. 8a and Fig. 8b show the cumulative distributions of TTFB for small and large objects, respectively. For small objects, we find that the TTFB distributions for C2DN and Replication (CDN) are similar, as expected: C2DN does not code these objects. C2DN slightly improves the TTFB distribution (shifting to the left) due to its lower object miss ratio. For large objects, we find about a 1 ms overhead in TTFB at low percentiles (25th-60th percentile). The slight increase is for cache hits due to fetching the first sub-chunk from K servers before serving the object.

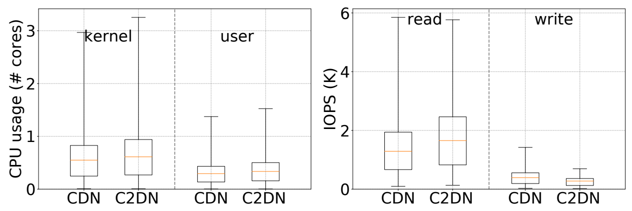
We now consider CDT. In practice, this metric is more relevant for large objects than the TTFB. Fig. 8c and Fig. 8d show a cumulative distribution of the content download time for small and large objects, respectively. Again we find that small objects behave similarly in Replication (CDN) and C2DN, with slightly better latency for C2DN due to lower object miss ratio. For large objects, C2DN and Replication (CDN) have a similar CDT. The overheads of fetching chunks are hidden by our streaming implementation based on sub-chunks (§5).

We remark that C2DN improves the tail latency in all cases (barely visible in the CDFs). For example, C2DN reduces the P90 TTFB by up to 3× compared to Replication (CDN). We attribute this to a lower miss ratio and the mitigation of stragglers using parity chunks to serve requests. This is as expected based on prior work on using coding to reduce tail latency [55].

6.6 Overhead assessment

We quantify the resource overheads of our C2DN prototype.

CPU usage. Fig. 9a measures CPU utilization in fractional CPU cores for userspace and kernel tasks, respectively. C2DN generally leads to higher CPU usage. The userspace CPU usage is higher due to the encoding and decoding of objects, and the kernel CPU usage is higher due to additional network and disk I/Os. Overall, CPU usage increases by 14% on average with a similar increase in kernel and userspace CPU usage.



(a) CPU usage **(b)** Disk IOPS

Figure 9: Resource usage. C2DN uses slightly more CPU resources and slightly more read disk IOPS than CDN, however, C2DN reduces write disk IOPS, especially at peak.

The increase in the CPU overhead is small as C2DN performs the encoding and decoding operation only on a fraction of requests. For the current coding size threshold of 128 KB, the number of requests served with coded objects is around 50%, while the number of bytes served using coded objects is close to 90%. Also, recall that most requests for coded objects do not need to be decoded as the object is recreated by concatenating data chunks in the output buffer. Decoding only happens in the case of stragglers and partial hits. In fact, only 6% of requests require decoding in our experiments. These cases happen primarily due to the straggler problem (individual slow servers); actual data chunk misses (partial hits) occur for less than 0.6% of requests. A future version of C2DN may further reduce CPU overheads by using kernel-bypass networking or increasing the object size threshold for coding. Increasing the threshold can happen with minimal side effects, as we show in the next section.

Disk usage. Fig. 9b compares disk IOPS of Replication (CDN) and C2DN for reads and writes. For reads, we observe that C2DN uses 23% more IOPS in the mean and less at the tail. Read-IOPS increase by 2% at the P99 and decrease by 11% at the P99.9 (we calculate this percentile across time and servers). For writes, C2DN uses 24% fewer writes IOPS in the mean. The tail write IOPS decreases by 46% at the P99 and 50% at the P99.9. The read IOPS increases because C2DN fetches at least $K = 3$ chunks to serve an object if coded. However, due to 1) most of the requests being for small uncoded objects, 2) the presence of DRAM hot object cache, the increase in reading IOPS is much smaller than 3×. While mean read IOPS increase, the peak read IOPS is similar or lower in C2DN. We attribute this to better load balancing in C2DN. Write IOPS in C2DN is significantly reduced when compared to Replication (CDN). C2DN has a lower storage overhead than Replication (CDN) and thus writes less to disk. In addition, the improvement in the miss ratio that C2DN provides further reduces the number of write operations. Besides, C2DN also improves the tail write IOPS, which is due to a better load balancing strategy of erasure coding and parity rebalance.

Intra-cluster network usage. C2DN uses network bandwidth within the cluster, about 0.9 Gbps in the mean and 2.3 Gbps at the P95. In conversations with CDN operators, this internal bandwidth usage is feasible for production clusters, as these links generally show little usage. For example, production CDN clusters use dedicated 10-Gbps-NICs for communication within the cluster.

6.7 Sensitivity analysis

We discuss the sensitivity of C2DN to its parameters.

Coding size threshold. The size threshold for coding impacts the performance in multiple ways. By reducing the size threshold, C2DN encodes more objects, improving cache space usage and load balance across cluster servers. At the same time, it leads to more CPU and I/Os (due to coding and fetching) and increases the latency for small objects. The size distribution in Fig. 1a shows that small objects contribute a small fraction of cache space usage. Thus, the potential benefit of coding diminishes as we decrease the size threshold for coding. At the same time, C2DN would use more cluster resources. We observe that reducing the size threshold to below 128 KB does not significantly benefit the object and byte miss ratio. Increasing the size threshold to over 8 MB increases the byte miss ratio by 2.79% and the write load imbalance by 258%. We believe that 128 KB is a good tradeoff for our production traces. Fig. 13 in the appendix shows our results.

Coding parameter K. Most of this section assumed C2DN configured with $K = 3$. We explore the impact of parameter K and P on miss ratio and write load balancing. We find that increasing K and keeping P constant reduces miss ratios for C2DN but increases miss ratios for C2DN-NoRebal. When adding chunks, the probability of getting partial hits increases for C2DN-NoRebal due to unbalanced eviction rates between servers. Because C2DN uses parity rebalance to achieve similar eviction rates between servers, the miss ratio decreases with increasing K due to lower storage overhead. While the impact of coding parameters has different impacts on miss ratios for C2DN-NoRebal and C2DN, the impact on load balancing is similar, as K increases, because an object is broken into more (and smaller) chunks, both the read and write load imbalance in C2DN-NoRebal and C2DN reduce. Fig. 14 in the appendix shows our results.

Different workloads. Throughout this section, we have used the video trace. We repeated our evaluation for the week-long web trace (§3). Compared to the video trace, the web trace has a significantly smaller working set. The video trace has a compulsory byte miss ratio of 0.1 and a compulsory object miss ratio of 0.21. In the web trace, the compulsory miss ratio is 0.06 for both byte and object miss ratios. In addition, compared to the video trace, the web trace has a more diverse object size range, as shown in Fig. 1a. Less than 10% of large objects (possibly large software) contribute to more than 90% of the cache space usage. Therefore, the fraction of requests that require coding is significantly smaller.

In prototype experiments with the web trace, only 3% of all requests are served coded. However, these 3% of requests account for 80% of served traffic. As a comparison, in the video trace, the prototype serves about 50% of requests from coded objects (with only 6% requiring decoding). Consequently, coding overheads on the web trace are negligible. In terms of the miss ratio, we observe a 10% reduction in object miss ratio and a 6% reduction in byte miss ratio. The write imbalance for Replication (CDN) is 1.72×, which is reduced to 1.03× in C2DN. The read imbalance for Replication (CDN) is 4.8×, which is reduced to 2.5× in C2DN.

Different eviction algorithms. Throughout this section, we have used FIFO as the eviction algorithm for the cache. FIFO provides stable performance on SSDs and extends the lifetime of an SSD by minimizing device write amplification [9,22,70]. Many open source caches such as Apache Trafficserver [7] and Varnish [71] use FIFO. To understand the impact of the eviction algorithm, we evaluate the Least-recently-used algorithm (LRU) using simulation. We observe a slight reduction in both object and byte miss ratios for all systems. All other results are qualitatively and quantitatively the same. Appendix 10.4, Fig. 16 and Fig. 17 show these results.

Variants of replication. Besides two-replication for all objects, CDNs have explored systems that replicate based on popularity. Specifically, only popular objects are replicated on two servers to reduce space overheads. As might be expected from our findings that write imbalance matters, popularity-based replication does not provide good fault tolerance. In simulation experiments, we observe object miss ratio spikes by 82%. Interestingly, we also observe that popularity-based replication leads to an even worse load imbalance than Replication (CDN), which explains the high miss ratio spike.

7 Discussion

DNS vs anycast-based CDN request routing. Different CDNs use different global load balancing architectures [33]. Akamai is well known for its DNS architecture [64]. Limelight [33], Wikipedia [58], and Cloudflare rely on anycast. While these designs have different performance implications, both rely on algorithms like consistent hashing. In DNS-based systems, consistent hashing is applied by the cluster-local load balancer to return the IP of the server responsible for the shard. Anycast-based systems typically route requests to any server in a cluster, and the server uses consistent hashing to identify another server that likely stores the object. Server unavailability, storage overheads of redundancy, and write imbalance are important problems in all CDN designs. While the cluster-local load balancer in our prototype relies on DNS, the principle design components of C2DN can be equally applied in anycast systems. We also expect that C2DN's benefits will transfer with similar quantitative improvements.

Larger clusters and multiple unavailabilities. In clusters of large size, multiple concurrent unavailabilities are not uncommon. As evaluated in §6, we find that C2DN is more effective in this setting as erasure coding is more efficient at tolerating multiple unavailabilities than replication. For large clusters, server unavailabilities become more common. We thus recommend either using a coding scheme with more parity chunks or handling the cluster as multiple smaller clusters.

8 Related work

While there is extensive work on caching, coding, load balancing, and flash caching, our work is uniquely positioned at the intersection of these areas. We discuss work by area.

Erasure coding in storage systems. Prior work has characterized the cost advantage offered by coding over replication in achieving data durability in distributed storage systems [75, 85]. Erasure codes are deployed in RAID [52], network-attached-storage [4], peer-to-peer storage systems [37, 40, 57, 79], in-memory key-value store [16, 17, 84], and distributed storage systems [32, 48, 56, 76]. Coding for CDNs differs due to the unique interplay of coding and caching and the two-sided transparency requirement (§4). Additionally, CDNs employ coding for different reasons (performance) than storage systems (durability), which magnifies overhead concerns.

Caching for coded file systems. Several recent works have explored augmenting erasure-coded storage systems with a cache to reduce latency [3, 29, 41, 55]. Aggarwal et al. [3] proposed augmenting erasure-coded disk-based storage systems with an in-memory cache at the proxy or the client-side that cache encoded chunks. Halalai et al. [29] propose augmenting geo-distributed erasure-coded storage systems by caching a fraction of the coded chunks in different geo-locations to alleviate the latency impact of fetching chunks from remote geo-locations. EC-Cache [55] employs erasure coding in the in-memory layer of a tiered distributed file system such as Alluxio (formerly [39]). Although EC-Cache is technically a cache, there is no interaction between coding and caching in EC-Cache since it operates in scenarios where the entire working set fits in memory, i.e., no evictions are considered. In contrast, C2DN focuses on CDN clusters with working sets in the hundreds of TB and starkly different tradeoffs, workload characteristics, and challenges as compared to file systems. In the area of cooperative caching [6, 30, 62], nodes synchronize caching decision via explicit communication. In contrast, C2DN proves that explicit communication is not required to synchronize the eviction of the K chunks, which significantly decreases overheads.

Chunking and caching. Prior work has explored the challenge of serving large files over HTTP, e.g., CoDeeN [74]. Similar to C2DN, CoDeeN breaks a large file into smaller chunks. A chunk cache miss does not require transferring the whole large file from the origin. In contrast to CodeeN, C2DN addresses unavailability tolerance, which is not provided by chunking alone.

Load balancing. Load balancing and sharding are well-studied topics [1, 2, 12, 13, 21, 27, 28]. To reduce the load imbalance, John et al. study the power of two choices that reduces the imbalance [12]. In addition, to serve skewed workloads, Fan et al. [24] study the effect of using a small and fast popularity-based cache to reduce load imbalance between different caches in a large backend pool.Yu-ju et al. [31] designed SPORE to use a self-adapting, popularity-based replication to mitigate load imbalance. Rashmi et al. [55]used erasure coding to reduce read load imbalance for large object in-memory cache. In summary, prior work on load balancing focuses on *read* load balancing, with little attention paid to *write* load balancing.

Load imbalance in consistent hashing can be solved with additional lookups via probing [47]. Unfortunately, these lookups are costly in CDNs (particularly for DNS-based systems). Additionally, this approach cannot be applied for erasure-coded caches due to the constraint that parity chunks should not be colocated with data chunks. In contrast to using load balancing to achieve a similar SSD replacement time, Mahesh et al. [8] used parity placement to achieve differential SSD ages so that SSDs of a disk array fail at different times.

Flash cache endurance. Flash caching is an active and challenging research area. A line of work [38, 45, 51, 63, 66, 67, 70] shows how eviction policies can be efficiently implemented on flash. Flashield [22] proposes to extend SSD lifetime via smart admission policies. All these systems focus on a single SSD. Our work focuses on wear-leveling across servers in a cluster, which significantly extends the lifetime of a cluster.

9 Conclusion

We re-architected the cluster of a CDN by introducing a hybrid redundancy scheme using erasure codes and replication, along with a novel approach for parity placement. We showed that our approach reduces the miss ratio and eliminates the miss ratio spikes caused by server unavailability. Further, our approach is more space-efficient than replication and is more attractive as CDN traffic and content footprint scale rapidly with Internet usage. Finally, our approach also reduces the write load imbalance by optimally placing the parities to reduce the lifetime of SSDs. We believe that C2DN is attractive for deployment in a production CDN since it integrates well with production CDN components.

Acknowledgements We thank our shepherd Angela Demke Brown and the anonymous reviewers for their valuable feedback. This work was supported in part by Facebook Fellowship, NSF grants CNS 1901410, CNS 1956271, CNS 1763617, and a AWS grant.

References

[1] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, Apr. 2010. USENIX Association.

[2] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016.

[3] V. Aggarwal, Y.-F. R. Chen, T. Lan, and Y. Xiang. Sprout: A functional caching approach to minimize service latency in erasure-coded storage. In *ICDCS*, 2016.

[4] M. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.

[5] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *Journal of the ACM (JACM)*, 18(1):80–93, 1971.

[6] S. Annapureddy, M. J. Freedman, and D. Mazieres. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 129–142. USENIX Association, 2005.

[7] Apache. Traffic Server, 2019. Available at https://trafficserver.apache.org/, accessed 09/18/19.

[8] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. Differential raid: Rethinking raid for ssd reliability. *ACM Transactions on Storage (TOS)*, 6(2):1–22, 2010.

[9] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, et al. The cachelib caching engine: Design and experiences at scale. In *USENIX OSDI*, pages 753–768, 2020.

[10] D. S. Berger, R. Sitaraman, and M. Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a cdn. In *USENIX NSDI*, pages 483–498, March 2017.

[11] M. Bjørling, J. Gonzalez, and P. Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, Feb. 2017. USENIX Association.

[12] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *International Workshop on Peer-to-Peer Systems*, pages 80–87. Springer, 2003.

[13] 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, New York, NY, USA, 2011. Association for Computing Machinery.

[14] D. G. Cattrysse and L. N. Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European journal of operational research*, 60(3):260–272, 1992.

[15] F. Chen, R. K. Sitaraman, and M. Torres. End-user mapping: Next generation request routing for content delivery. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 167–181. ACM, 2015.

[16] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)*, 13(3):1–30, 2017.

[17] L. Cheng, Y. Hu, and P. P. C. Lee. Coupling decentralized key-value stores with erasure coding. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 377–389, New York, NY, USA, 2019. Association for Computing Machinery.

[18] B. V. Cherkassky and A. V. Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[19] CISCO. Global IP traffic forecast: The zettabyte era—trends and analysis, June 2017. Available at https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.pdf, accessed 24/09/17.

[20] J. Dilley, B. M. Maggs, J. Parikh, H. Prokop, R. K. Sitaraman, and W. E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[21] 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 *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[22] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked*

Systems Design and Implementation (NSDI 19), pages 65–78, Boston, MA, Feb. 2019. USENIX Association.

[23] R. Fagin. Asymptotic miss ratios over independent references. *Journal of Computer and System Sciences*, 14(2):222–250, 1977.

[24] 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*, pages 1–12, 2011.

[25] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[26] C. Fricker, P. Robert, and J. Roberts. A versatile and accurate approximation for LRU cache performance. In *ITC*, page 8, 2012.

[27] 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. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.

[28] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. Basil: Automated io load balancing across storage devices. In *Fast*, volume 10, pages 13–13, 2010.

[29] R. Halalai, P. Felber, A.-M. Kermarrec, and F. Taïani. Agar: A caching system for erasure-coded data. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 23–33. IEEE, 2017.

[30] V. Holmedahl, B. Smith, and T. Yang. Cooperative caching of dynamic content on a distributed web server. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No. 98TB100244)*, pages 243–250. IEEE, 1998.

[31] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–17, 2013.

[32] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[33] C. Huang, A. Wang, J. Li, and K. W. Ross. Measuring and evaluating large-scale cdns. In *ACM IMC*, volume 8, pages 15–29, 2008.

[34] B. Jiang, P. Nain, and D. Towsley. On the convergence of the ttl approximation for an lru cache under independent stationary request processes. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(4), Sept. 2018.

[35] S. Kadekodi, F. Maturana, S. J. Subramanya, J. Yang, K. Rashmi, and G. R. Ganger. {PACEMAKER}: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 369–385, 2020.

[36] 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*, pages 654–663, 1997.

[37] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[38] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage (TOS)*, 13(3):1–34, 2017.

[39] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SoCC*, 2014.

[40] J. Li and C. Zhang. Distributed hosting of web content with erasure coding and unequal weight assignment. In *2004 IEEE International Conference on Multimedia and Expo (ICME) (IEEE Cat. No.04TH8763)*, volume 3, pages 2087–2090 Vol.3, 2004.

[41] K. Liu, J. Peng, J. Wang, and J. Pan. Optimal caching for low latency in distributed coded storage systems. *arXiv preprint arXiv:2012.03005*, 2020.

[42] R. S. Liu, C. L. Yang, C. H. Li, and G. Y. Chen. Duracache: A durable ssd cache using mlc nand flash. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–6, 2013.

[43] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.*, 45(3):52–66, July 2015.

[44] V. Martina, M. Garetto, and E. Leonardi. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM*, 2014.

[45] S. McAllister, B. Berg, J. Tutuncu-Macias, J. Yang, S. Gunasekar, J. Lu, D. S. Berger, N. Beckmann, and G. R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[46] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407, 2018.

[47] V. Mirrokni, M. Thorup, and M. Zadimoghaddam. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 587–604, 2018.

[48] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm BLOB storage system. In *USENIX OSDI*, pages 383–398, 2014.

[49] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at Facebook. In *USENIX NSDI*, pages 385–398, 2013.

[50] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[51] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, volume 12, 2012.

[52] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.

[53] L. Perron and V. Furnon. OR-tools. https://developers.google.com/optimization/.

[54] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *SIGCOMM*, 2015.

[55] 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*, pages 401–417, 2016.

[56] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.

[57] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *FAST*, 2003.

[58] E. Rocca. Running Wikipedia.org, June 2016. available https://www.mediawiki.org/wiki/File:WMF_Traffic_Varnishcon_2016.pdf accessed 09/12/16.

[59] R. K. Sahoo, M. S. Squillante, A. Sivasubramaniam, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *International Conference on Dependable Systems and Networks, 2004*, pages 772–781, 2004.

[60] L. Saino, I. Psaras, and G. Pavlou. Understanding sharded caching systems. In *IEEE INFOCOM*, pages 1–9, 2016.

[61] Sanjay Sane. Latency and wear-out in facebook's cdn due to ssd write pressure. Private conversation,, 7 2019.

[62] P. Sarkar and J. H. Hartman. Hint-based cooperative caching. *ACM Transactions on Computer Systems (TOCS)*, 18(4):387–419, 2000.

[63] M. Saxena, M. M. Swift, and Y. Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 267–280, New York, NY, USA, 2012. Association for Computing Machinery.

[64] K. Schomp, O. Bhardwaj, E. Kurdoglu, M. Muhaimen, and R. K. Sitaraman. Akamai DNS: Providing authoritative answers to the world's queries. In *ACM SIGCOMM*, pages 465–478, 2020.

[65] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing*, 7(4):337–350, 2009.

[66] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Optimizing flash-based key-value cache systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[67] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Didacache: an integration of device and application for flash-based key-value caching. *ACM Transactions on Storage (TOS)*, 14(3):1–32, 2018.

[68] A. Sundarrajan, M. Feng, M. Kasbekar, and R. K. Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *ACM CoNEXT*, pages 55–67, 2017.

[69] A. Sundarrajan, M. Kasbekar, R. K. Sitaraman, and S. Shukla. Midgress-aware traffic provisioning for content delivery. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 543–557, 2020.

[70] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. Ripq: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[71] F. Velázquez, K. Lyngstøl, T. Fog Heen, and J. Renard. *The Varnish Book for Varnish 4.0*. Varnish Software AS, March 2016.

[72] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, 2017.

[73] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient mrc construction with shards. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, 2015.

[74] L. Wang, K. Park, R. Pang, V. S. Pai, and L. L. Peterson. Reliability and security in the codeen content distribution network. In *USENIX ATC*, pages 171–184, 2004.

[75] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS*, 2002.

[76] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[77] G. Wu and X. He. Reducing ssd read latency via nand flash program and erase suspension. In *FAST*, volume 12, pages 10–10, 2012.

[78] K. Wu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Towards an unwritten contract of intel optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[79] F. Xu, Y. Wang, and X. Ma. Online encoding for erasure-coded distributed storage systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 338–342, 2017.

[80] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Trans. Storage*, 13(3), Oct. 2017.

[81] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, Nov. 2020.

[82] J. Yang, Y. Yue, and K. V. Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Trans. Storage*, 17(3), aug 2021.

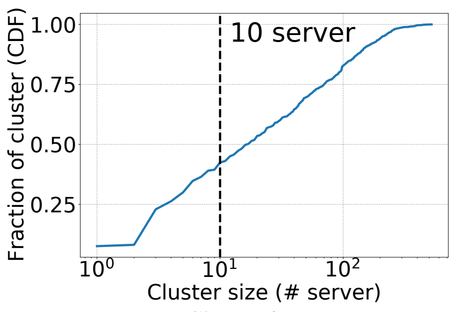
[83] J. Yang, Y. Yue, and R. Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, Apr. 2021.

[84] M. M. T. Yiu, H. H. W. Chan, and P. P. C. Lee. Erasure coding for small objects in in-memory kv storage. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, New York, NY, USA, 2017. Association for Computing Machinery.

[85] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does erasure coding have a role to play in my data center? Technical Report Microsoft Research MSR-TR-2010, 2010.

10 Supplemental information

10.1 Cluster size distribution



(a) Cluster size

Figure 10: Cluster size ranges from 1 to over 500 servers.

10.2 Proof details

Proof of Theorem 1.
Let T_c^i denote the *characteristic time* [23, 26] of the cache at server i, given capacity C, where the characteristic time of an LRU cache measures how long it takes for a newly requested chunk to get evicted. We first prove that for any two servers i and j, T_c^i and T_c^j are nearly the same. More precisely, we show that for any $i \neq j$, $Prob(|T_c^i - T_c^j| \geq \varepsilon)$ is at most $O(W/(\varepsilon^2 C))$, using a mathematical argument similar to [60]. Where C denotes the cache size of each server and W is the variance of the write load imbalance across the servers in the cluster. Due to parity rebalancing in C2DN, $W \rightarrow 0$. So, this probability $O(W/(\varepsilon^2 C))$ vanishes as the cache size grows large.

In C2DN, when an object is requested, its chunks $x_1, \dots, x_n$ are requested at the same time from the individual servers. Since the characteristic time of the servers that these chunks reside in are (nearly) the same as shown above, it follows that these chunks are evicted from these caches at (nearly) the same time. Thus, the chunks $x_1, \dots, x_n$ of an object enter and exit their individual caches in a synchronized way, even though there is no explicit coordination among the caches.

10.3 Additional details on parity rebalance

Here, we give more details about the bucket assignment algorithm discussed in §4 under the three scenarios (1) Initial bucket assignment, (2) Server failure, (3) Server addition. We first consider the case where the servers are homogeneous and later extend the algorithm to the heterogeneous case.

Initial bucket assignment. The algorithm runs in two phases. In the first phase, the data buckets are assigned to the servers using the consistent hashing algorithm. The algorithm chooses K consecutive servers on the consistent hash ring from the bucket's hash location in a clockwise direction to assign the K data chunk buckets. In the second phase, the parity chunks are assigned to the servers such that the load is evenly balanced across the servers. The second phase is described in Algorithm 1.

Algorithm 1 Phase 2. Parity rebalance

```
1: Input : Set of available servers $\mathcal{A}$ and the total write load on the cluster $W$.
2: Set of $\mathcal{N}$ parity buckets. For $n \in \mathcal{N}$, the sum of sizes of the parity chunks in the bucket is $s_n$.
3: A set $L$ with $l_i$ denoting the current write load (due to assignment of data buckets and uncoded objects) on server $i$.
4: Output : A valid assignment of parity bucket to the servers.
5: Initialize : A set of vertices $V \leftarrow \phi$, a set of edges $E \leftarrow \phi$, an empty graph $\mathcal{G} = (V, E)$.
6: Add source node $S$, terminal node $T$, nodes corresponding to parity buckets and available servers to $V$.

7: for $n \in \mathcal{N}$ do // Loop through parity buckets
8:     $e \leftarrow ((S,n), n_s)$ // Create edge between nodes between source-node $S$ and bucket-node $n$ with weight equal to size of the bucket $n_s$
9:     $V \leftarrow n$, $E \leftarrow e$
10:    for $a \in \mathcal{A}$ do
11:        if data chunk of bucket $n$ is not assigned to $a$ then
12:            $e \leftarrow ((n,a), n_s)$ // Create edge between parity bucket node $n$ and available server $a$ with size of parity bucket $n_s$.
13:        end if
14:    end for
15: end for

16: for $a \in \mathcal{A}$ do // Loop through available servers
17:     $c_a \leftarrow max(\lceil \frac{W}{|\mathcal{A}|} \rceil - l_i, 0)$ // Available budget on each server
18:     $e \leftarrow ((a,T), c_a)$ // Create edge between server nodes $a$ and sink-node $T$ with weight $c_a$
19:     $V \leftarrow v$, $E \leftarrow e$
20: end for

21: Run MaxFlow($S$, $T$) between the source-node $S$ and terminal node $T$.
22: for Each parity bucket $n \in \mathcal{N}$ do
23:     Assign the parity bucket to the least loaded server that has a positive assigned flow from the bucket.
24: end for
```

Post running the Phase 1 of the algorithm, the available write budget on each server a (c_a) in line 17 of Algorithm 1, is obtained as follows. If the total unique bytes requested to the cluster is W, then each server should host traffic not more than $\left\lceil \frac{W}{|\mathcal{A}|} \right\rceil$ bytes. After phase 1 of the algorithm, if l_i is the traffic assigned to server i, then the available budget on server i i.e., c_i is obtained as $\max(\left\lceil \frac{W}{|\mathcal{A}|} \right\rceil - l_i, 0)$. An empty graph is initialized in line 5 and the source and terminal nodes are added in line 6. In line 8, for each bucket n we add an edge from the source node S to the bucket-node with a capacity that equals the size of the parity bucket. Through lines 10-14, for each bucket n add a directed edge from the node that corresponds to the parity bucket to a server-node that is not assigned a data chunk of bucket n. These edges capture if the parity bucket can be assigned to a server. Then we assign these edges with a capacity that equals to the size of the parity bucket. In lines 16-19 directed edges are added from server nodes (a) to the sink node with a capacity c_a i.e., the available write budget on the server. Now run the max-flow algorithm in the graph between the source-node S and the sink-node T to find a valid assignment of parity buckets to the servers. To assign a parity bucket to a server, we find edges from the parity-node to the server-nodes that are assigned a positive flow. The servers are potential candidates for assignment. Empirically, we find that in most runs, the algorithm finds a single candidate server, if not, we assign the parity bucket to the least loaded server among the potential candidates.

Server failure. When a server fails the data buckets (D) and parity buckets (P) belonging to the server need to be reassigned. As done previously, the data buckets are reassigned using the consistent hash ring. Now, the new data bucket allocation could invalidate some of the previous parity bucket assignments (on the currently available servers) as the data chunks and parity chunks cannot cohabit the same server. Let the invalidated parity buckets be P'. The available budget c_i of each server i is recalculated using the total traffic W and the number of available servers $|\mathcal{A}|$. Now, the buckets $P \cup P'$ (parity buckets of failed server and invalidated parity buckets) are assigned to the servers using Algorithm 1 by reassigning corresponding capacities in line 9. When a server fails we also keep track of the parity buckets that were assigned to it before failure. Some of these buckets could be reassigned to the server when it is available again depending on the available write budget at each server.

Server addition. When a server is available again, it gets assigned the data buckets using the consistent hashing algorithm. We recompute the capacity of each server as done previously. Now, as we have kept track of the parity buckets the server was assigned prior to failure, we try to re-assign as many of those parity buckets as possible. If we get back all buckets and we still have capacity for more buckets to be assigned, we iteratively pick parity buckets from the most loaded server.

Algorithm complexity. In Algorithm 1 the cost of constructing the graph can be computed using $O(|T| \times |\mathcal{N}|)$ time complexity where T is the number of servers. And the time complexity is because we need to decide if we wish to add an edge between the parity chunk and the server. Further, the runtime complexity of the MaxFlow algorithm in line 21 is computed as follows. The total available budget on the servers is $B = \sum_{i \in S} \left(\left\lceil \frac{W}{|\mathcal{A}|} \right\rceil - l_i \right)$. Then, the runtime complexity is $B \times |\mathcal{N}| \times |T|)$.

Heterogeneous servers. We extend consistent-hashing-based bucket assignment to the case of heterogeneous servers. Each server is represented by at least one virtual node on the consistent hashing ring. The number of virtual nodes added is proportional to the capacity of the server (e.g., if server A is 2× larger than server B, then server A would have 2× as many virtual nodes).

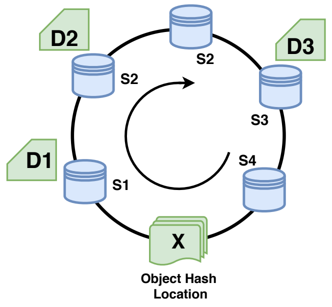
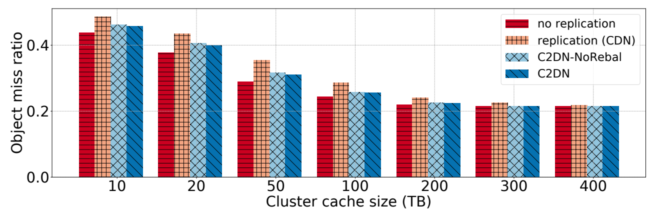


Figure 11: Consistent Hashing for a cluster with heterogeneous servers. The data buckets D1, D2 and D3 are hashed to unique servers S1, S2 and S3.

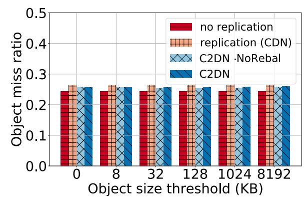
The data buckets are mapped to the consistent hashing ring as follows. From the bucket's hashed position on the ring, we move along the ring in a clockwise direction and assign — the K data buckets — iteratively to the virtual nodes encountered. However, while assigning, we step over servers that have been assigned any of the other K data buckets. This ensures each of the K data buckets are assigned to different servers. Figure 11 shows a placement example. The cluster consists of 4 servers S1, S2, S3 and S4 of capacity C, 2C, C, C respectively. Virtual nodes are indicated by the server name. Note that, as S2 is twice the capacity of the servers, we create two virtual nodes for S2. By chance, we assume that the two virtual nodes hash to adjacent positions on the ring. Now, if the bucket is hashed to a location X on the consistent hash ring, then the data buckets D1, D2 and D3 are assigned to the first three servers encountered by moving in the clockwise direction from X. This is S1, S2 (S2 again and thus skipped), and S3, which is the resulting bucket placement. After data buckets are assigned, we use Algorithm 1 to assign parity buckets.



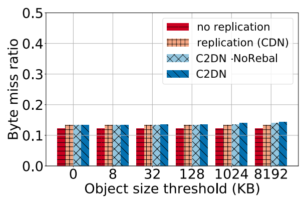
(a) Object miss ratio

Figure 12: Object miss ratio of different systems.

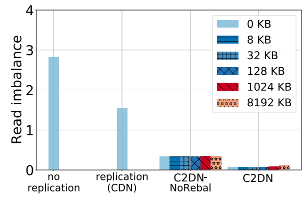
10.4 Additional figures for sensitivity analysis



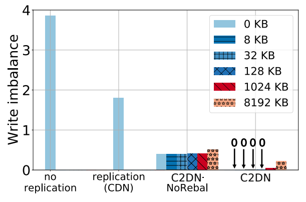
(a) Object miss ratio



(b) Byte miss ratio

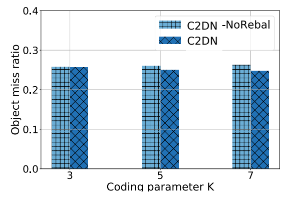


(c) Read load balancing

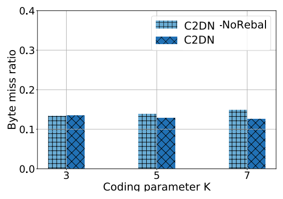


(d) Write load balancing

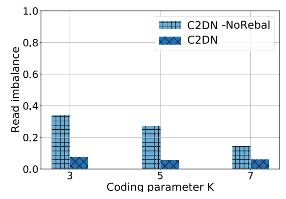
Figure 13: Impact of coding size threshold on miss ratio and load balancing.



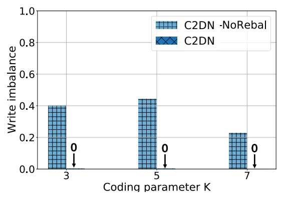
(a) Impact on object miss ratio



(b) Impact on byte miss ratio

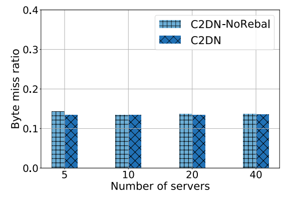


(c) Impact on read imbalance

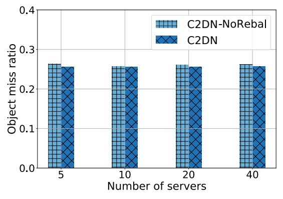


(d) Impact on write imbalance

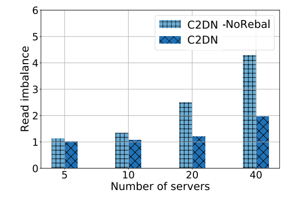
Figure 14: Impact of parameter K on miss ratio and load imbalance.



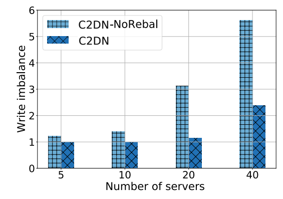
(a) Impact on byte miss ratio



(b) Impact on object miss ratio

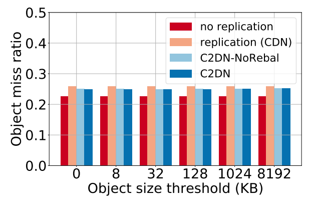


(c) Impact on read imbalance

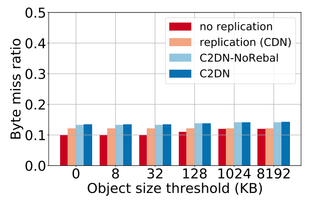


(d) Impact on write imbalance

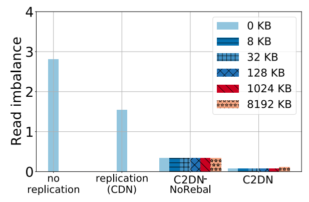
Figure 15: Impact of number of servers on miss ratio and load imbalance.



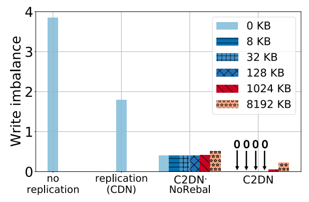
(a) Object miss ratio



(b) Byte miss ratio

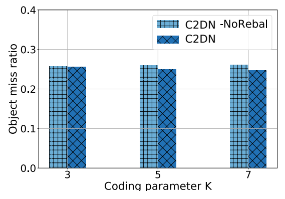


(c) Read load balancing

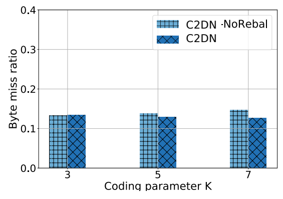


(d) Write load balancing

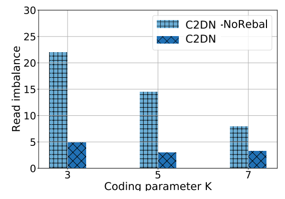
Figure 16: Impact of coding size threshold on miss ratio and load balancing (LRU).



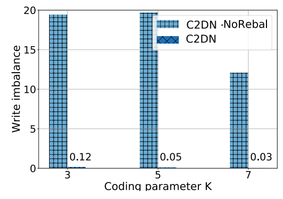
(a) Impact on object miss ratio



(b) Impact on byte miss ratio



(c) Impact on read imbalance



(d) Impact on write imbalance

Figure 17: Impact of parameter K on miss ratio and load imbalance (LRU).