Characterizing Load Imbalance in Real-World Networked Caches

Qi Huang[†‡], Helga Gudmundsdottir[§ß], Ymir Vigfusson[§ß], Daniel A. Freedman[ϧ]
Ken Birman[†], and Robbert van Renesse[†]
[†]Cornell University, [‡]Facebook Inc., [§]Emory University, [ß]Reykjavik University, [ϧ]Technion – Israel Institute of Technology

Abstract

Modern Web services rely extensively upon a tier of in-memory caches to reduce request latencies and alleviate load on backend servers. Within a given cache, items are typically partitioned across cache servers via consistent hashing, with the goal of balancing the number of items maintained by each cache server. Effects of consistent hashing vary by associated hashing function and partitioning ratio. Most real-world workloads are also skewed, with some items significantly more popular than others. Inefficiency in addressing both issues can create an imbalance in cache-server loads.

We analyze the degree of observed load imbalance, focusing on read-only traffic against Facebook's graph cache tier in Tao. We investigate the principal causes of load imbalance, including data co-location, non-ideal hashing scenarios, and hot-spot temporal effects. We also employ trace-drive analytics to study the benefits and limitations of current load-balancing methods, suggesting areas for future research.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]:Distributed Applications; H.3.4 [**Systems and Software**]:Performance Evaluation

General Terms

Measurement, Experimentation

Keywords

Caching, Load Balancing, Networking

1 Introduction

In-memory caches are often used in modern Web services to decrease request latency for users, as well as to relieve load on storage and database servers. Instead of executing a potentially resource-intensive operation on a backend server directly, Web front-ends first consult an appropriate cache server for a copy of the desired data.

The raw aggregate request volume at popular websites would significantly overwhelm the capacity of a single cache server. As such, data is normally divided among hundreds or thousands of cache servers [1, 5], typically by partitioning the large space of possible data-object IDs into segments that are then mapped onto cache servers. The segments — called *shards* — commonly contain a large number of objects to reduce the size of the object-to-server lookup map. Further, "related" objects (*i.e.*, those benefiting from co-location on the same server) tend to be mapped to the same shard to mitigate the impact of *thundering herds* [13] and decrease query fan-out [1].

Ideally, cache servers would all observe similar request rates (volume per unit time), since this would provide predictable request latencies [6] and reduce the over-provisioning of resources necessary to withstand peak workloads [8, 15]. In reality, however, segments confronted with real-world workloads often sustain variable and dynamic request rates that can contribute to significant load imbalance. Imbalance can be caused by the skewed access-popularity among different objects, as well as the decision to co-locate related data within the same segment to support more advanced data queries. While the skewed popularity applies for most Web sites traffic, data co-location is often adopted for structural data (*e.g.* Facebook's social graph data.)

There are many options for balancing the number of objects apportioned to distributed cache servers. Most mechanisms randomly partition data across servers by hashing [10, 11], while some additionally adapt to changes [4, 8]. Much recent interest has also focused on understanding and mitigating hot spots and load imbalance that arise in skewed workloads seen in key-value stores and cache systems [5, 6], but, to the best of our knowledge, no comprehensive analysis to date permits online analysis of the key culprits based on a real workload.

In this paper, we investigate the nature of load imbalance based on the real-world setting of Facebook's social-graph cache, Tao, where data have skewed access popularity and cannot be randomly separated. We analyze the resulted imbalance from each factor, explore how different categories of approaches — fine-tuned consistent hashing and hot-content replication (including the special case for front-end caching) — might help to mitigate the impact of these factors, and identify limitation of such approach categories.

Our paper offers the following contributions:

- We identify that the popularity skewness at object level is not a major cause of cache load-imbalance.

HotNets '14, October 27–28, 2014, Los Angeles, CA, USA

ACM 978-1-4503-3256-9/14/10...$15.00
http://dx.doi.org/10.1145/2670518.2673882

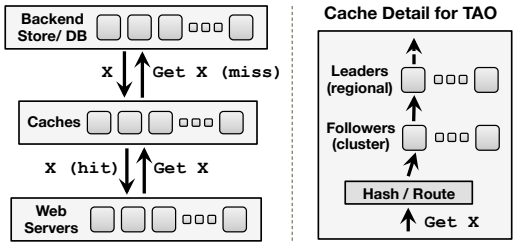

Figure 1: *Cache architecture: (L) Operation of read-through cache between front-end Web servers and back-end data; (R) Organization of Facebook's* Tao *cache.*

- We deduce that load imbalance in systems like Tao can stem from a combination of load-insensitive partitioning, extremely hot shards, and random temporal effects.
- We survey current approaches to load balancing and, through simulation, assess their effectiveness on our real-world traces in order to guide future research.

2 Analyzing Load Imbalance

In order to properly justify load imbalance within real-world systems, we study in-memory cache-access traces from Facebook's graph-storage, Tao. Given that Tao's general design shares commonality with other caching solutions, our discoveries in Tao apply to the broader domain.

2.1 Environment

Figure 1 illustrates a typical architecture of in-memory caches that facilitate the modern Web stack. When a front-end Web server receives clients' HTTP requests, it often spawns numerous data-fetching requests to generate a content-rich response. To reduce the fetching latency and database-querying overhead, such data fetches are first directed to a tier of caches, where popular content results reside in DRAM. A data request reaches the backend of the stack only if the requested data is not found in the caching space. Based on the manner in which the backend request is redirected, there are two categories of caching tiers: read-through and read-aside. The request flow in Figure 1 follows the read-through style, in which the cache serves as a proxy to fetch data from the backend while interfacing with the Web server. In the read-aside style, the Web server is responsible for requesting the data from backend servers and subsequently inserting the content into the cache. Facebook's graph-storage solution Tao organizes its cache tier as read-through caches, specifically with two layers of caches: follower and leader. The follower cluster co-locates with each Web front-end cluster, while a leader cluster supports all follower clusters for an entire region[1].

Sharding. Sharding or partitioning is a general approach to scale a database beyond any single server's capacity and is commonly used in production caching systems [1, 13]. Sharding can be achieved via two methods, based upon the requirement for co-locating related data — *random sharding* and *dependent sharding*. In a normal key-value interface system, such as `memcached`, random sharding is sufficient for distributing relatively similar numbers of objects to each shard, and the independence of objects within the same shard does not impact the system's GET/SET operation performance. However, dependent sharding is more appealing for systems that provide advanced queries based on structural data. For instance, Tao enables range queries on its social graph, constituting almost 44% [1] of its operations. By using dependent sharding to co-locate socially-connected graph objects, Tao is able to reduce the range-query fan-out and associated network overhead. Moreover, dependent sharding also provides the possibility of consistency tracking on related data. Our analysis results in this paper are more relevant to these dependent-sharding caches, where the access disparity between different shards may be more profound than the disparity of different cached items. Figure 2 shows an example of dependent sharding for graph. Ignoring the sharding type, shard-to-server mapping is often conducted through consistent hashing.

[1]Region stands for a geographical location where one or multiple data centers are located.

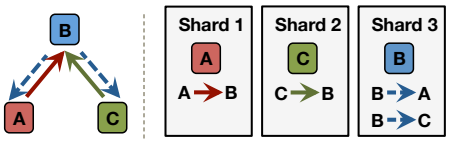

Figure 2: *Dependent sharding (R) on directional graph (L): edges are co-located with source vertices within same shard.*

Trace	Sampling Metric	Sampling Ratio	Details
TopShards	Time	1 min	Top 100 shards, 20 objs load per cache
ReqSample	Request	10^5 reqs	Requests sent to cache cluster
ServerRep	Time	4 min	Reported server load in req/sec

Table 1: Tao *trace summary:* **TopShards** *reports hot-content traffic on each cache;* **ReqSample** *samples requests from Web servers;* **ServerRep** *contains cache-load snapshots.*

Traces. Throughout this paper, we rely upon three sets of Tao production traces, collected between Facebook's front-end Web servers and Tao followers. Table 1 describes each trace, and their usage within this paper is explained below.

- **TopShards** is directly reported by each cache server, constituting the exact number of requests sent to the 100 most popular shards and 20 most popular objects per server. This trace gives us a high-quality source to analyze the impact of load imbalance due to popularity skewness and the temporal dynamics of popular objects and shards.
- **ReqSample** is sampled among the real read requests sent from Web servers. While the per-request sampling gives a less-detailed signal for a temporal analysis on popular objects and shards, the full fidelity of the request flow provides a solid basis for simulation.

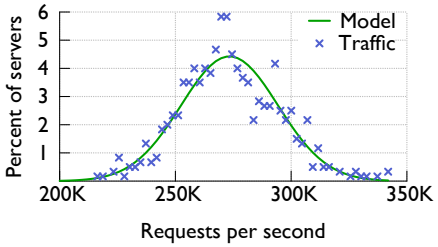

Figure 3: *Load distribution (a cluster).*

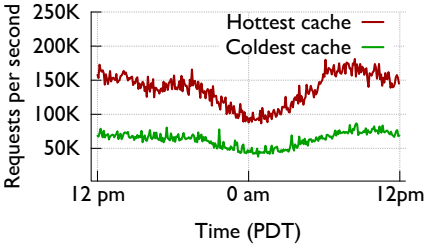

Figure 4: *Load disparity within a day.*

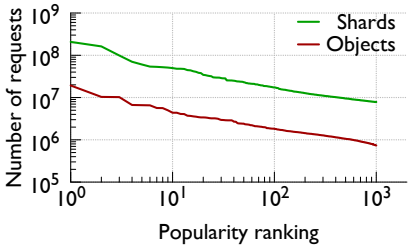

Figure 5: *Content popularity.*

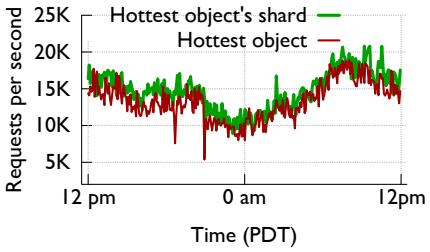

Figure 6: *Traffic for hottest object.*

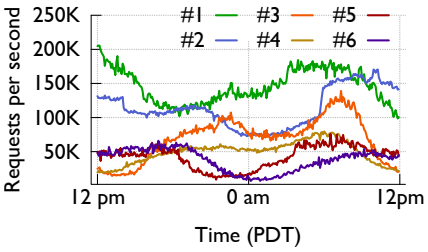

Figure 7: *Traffic for hottest shards.*

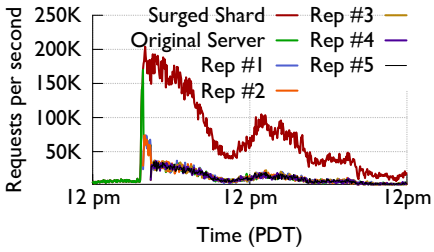

Figure 8: Tao *reaction for surged shard.*

- **ServerRep** is a per-cache traffic-rate report collected every four minutes, as an adjunct to the **TopShards**. Thus, **ServerRep** reports the entire cache traffic, instead of only that of the hot contents. It serves as ground truth in our study, validating the cache-load status from the **TopShards**-driven analysis and the **ReqSample**-driven simulation.

2.2 Analysis

Though Tao already deploys several load-balancing techniques [1], it still experiences a certain degree of load imbalance within the caching tier. Figure 3 quantifies this over a 24-hour period, within a follower cluster from **TopShards**, by revealing the statistical profile of load as a normal distribution, with a mean of $247K \pm 21K$ requests per second ($p < 0.05$, $A = 0.7767$, Anderson-Darling normality test). **ServerRep** confirms this result with full cache traffic. Figure 4 further illustrates the traffic dynamics of both the hottest and coldest servers from the same cluster. As can be seen, the disparity exists throughout the day, as both curves follow similar diurnal patterns, though the contrast is greater at peak hours (>100K requests per second [rps] difference) and smaller during the idle period (<50K rps difference). In order to find the root causes of this, we primarily examined traces to answer the following questions:

- Does skewed content popularity impact load disparity in the cache?
- Does imbalanced placement of similarly popular content play a major role?

Skewed content popularity. We used our traces to confirm that content popularity, defined in terms of number of accesses, resembles a power-law distribution across two different populations: distinct objects (vertex and edge in the social graph), and different shards (each of which maintains some partition of the graph). Figure 5 shows the number of requests for each of the top-1000 objects and top-1000 shards, as reported in **TopShards**. In light of this, does the hottest object become a dominant resource bottleneck for a cache server? Figure 6 shows the traffic dynamics of the most popular object and its associated shard. We note that a single hot object can contribute almost the entire traffic for its hosting shard, sufficient to push the shard into the top-100. However, notice the level of imbalance: the traffic associated with this hot object wouldn't even be half the available capacity for the coldest cache server (Figure 4). Accordingly, a skewed popularity at object level does not signify a major cause of cache-load imbalance for the request traffic between front-end Web servers and cache servers. As discussed further in Section 3, this explains the limited benefit in balancing cache load via an additional layer of front-end cache.

The status is different for shard-level traffic. Figure 7 examines the traffic dynamics of the six top-ranked shards, based on their popularity. Compared to object-traffic (Figure 6), popular shards receive significantly higher load, due to the combination of many related objects; these shards have much greater impact on a cache server. It is noteworthy that each curve in Figure 7 combines multiple reports from servers that all hold replicas of the same shard.

Our question is made more complex because Tao itself has an architectural feature that comes to bear here. Within Tao, shard replication is such that a hot shard will be spread over multiple cache servers, and the front-end router redirects Web-server requests among them. Specifically, the hottest shard has been replicated 10-fold throughout the 24-hour period. Such replication starts when a server has less than 20% remaining CPU- and network- capacity, while more than

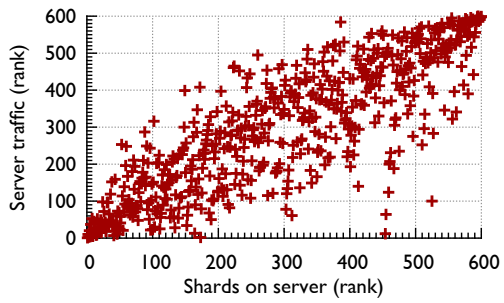

Figure 9: *Load imbalance impact from shard placement.*

25% of the request load comes from a "dominant" shard. In other words, as long as a shard contributes 20% of a busy server's capacity cap it needs to be replicated. From our analysis, Tao replication plays a significant role in keeping servers' loads below 200K rps.

Some aspects of the existing replication mechanism need improvement: **(1)** if a shard becomes hot too quickly, replication is too slow to react, and **(2)** the current reverse routine (de-allocating server space for no-longer-hot shards) is too conservative. Figure 8 shows 48 hours of traffic dynamics for a "popularity-surged shard," depicting the involvement of up to five extra replicas to split the request load. When the traffic to the popularity-surged shard rises from <75K rps to >175K rps within ten minutes, the replication process starts (but only after the original server stays at that load for another eight minutes); replication continues an hour later when the shard is still causing too much load on its replicas. Moreover, once all five replicas are created, they remain so situated for another 14 hours — even as the shard's popularity plummets to <50K rps, wasting significant cache memory. In Section 3, we further discuss the benefits and limitations of replication techniques in general and propose a potential improvement.

Problematic content placement. While consistent hashing is used in Tao to balance the mapping between data partitions (shards) and servers, studies of consistent hashing in other settings suggest that often the implementation of this mechanism is not sophisticated enough to overcome intrinsic issues with consistent hashing: insufficient rounds of hashing, low ratios between shards and servers, and poor choices of hashing functions. As a result, the shards may not be evenly distributed among different servers. To investigate the status of content placement in Tao and possible impact on load imbalance, we examined the correlation between the number of shards hosted by a server and the total traffic it serves.

Figure 9 shows that there is a strong relationship between the rank of cache servers by number of shards and their rank by traffic load (Spearman's $\rho = 0.848$, $p < 10^{-5}$). This correlation is especially true in the extreme cases: servers hosting the most shards tend to rank among the most-loaded ones, and servers hosting the fewest shards rank among those least-loaded. This demonstrates that, within the Tao system, non-ideal shard placement plays an important role, alongside skewed content popularity, in causing load imbalance at the granularity of shards.

3 Mitigating Load Imbalance

Using simulations on Tao traces, we now evaluate two major categories of techniques that seek to mitigate load imbalance: **(1)** consistent hashing; and **(2)** hot-content replication, which also includes front-end caching as a special case.

3.1 Trace Preparation

In order to properly evaluate traffic dynamics under different load-balancing approaches, we favor our **ReqSample** trace, due to its strong fidelity of actual cache requests. The original trace is collected by the routing daemon, mcrouter [12], on every Web front-end server; mcrouter randomly samples and records one out of every million Tao requests (so as to minimize measurement overhead on Facebook's live infrastructure). Cross-validation shows that the trace successfully captures all hot shards and hot objects contained in the **TopShards** aggregate trace. However, **ReqSample**'s sampling is too coarse to retain the traffic characteristics of every shard within a single cluster.

To cope with this problem, we treat this entire **ReqSample** trace as a trace for a single "canonical silo" cluster that has been sampled at a higher frequency. This is feasible because **(1)** every Tao follower cluster is an independent caching deployment; and **(2)** graph queries to each Tao cluster behave similarly for popular content since Web requests are randomly distributed among all front-end clusters.

We verified these two properties on the **TopShards** by comparing the traffic dynamics of the top 1000 shards between a single cluster and the entire tier. The full-tier trace in **ReqSample** is effectively an aggregate of multiple clusters serving the same content (in different regions). Hence, we normalize the traffic to our canonical silo cluster based on that of one of the largest single clusters in the full trace. The manipulations on the entire trace yield the same normalized load distribution on the canonical-silo cluster as originally found ($p < 10^{-5}$, $D = 0.2883$, Kolmogorov-Smirnoff test), except with higher sampling frequency.

3.2 Current Techniques

Our goals are twofold: **(1)** to understand how state-of-the-art approaches, for mitigating load skew on a distributed cache, complement one another on a real-world trace; and **(2)** to identify which mechanisms suggest opportunities for improvement. Two main classes of algorithms work in tandem to balance load: hashing schemes for balancing the *number* of shards allocated to servers, and replication schemes for balancing the *load* of these shards.

Hashing. The concept supporting most partitioned services is that hashing shard identifiers to an abstract ring, and then dividing segments of this ring among servers, will yield roughly fair assignment of shards to servers. Assuming the ring state is maintained in an up-to-date status on every server, all lookups

may be done locally — a property which facilitates distributed implementation. As noted earlier, many existing systems leverage *consistent hashing* [10] which minimizes disruptions when servers are added or removed. Our experiments here include the popular open-source `libketama` library, a reference implementation of consistent hashing for in-memory networked caches [9].

Unfortunately, hashing schemes may impose significant skew on the load distribution. This is partially explained by consistent-hashing's disregard of traffic on shards. However, even if all shards carry the same volume of traffic, the server with the highest load would still — with high probability — be responsible for twice as many shards as an average server [14].

Better distributed hashing schemes may yet be found. One existing remedy for this uneven division of shards is to further divide the ring space by hashing each server identifier many times onto the ring as "virtual nodes" [3, 4, 11]. Recently, Hwang and Wood proposed an adaptive hashing mechanism based on consistent hashing, where the segments boundaries of the ring space are dynamically adjusted according to load and cache hit rate on the servers to which they are assigned [8]. However, one must still address the problem of disparate traffic rates on shards.

To understand the opportunity for improving hashing mechanisms, we include a "*perfect hashing*" baseline in our simulations. In this baseline, a centralized controller ensures that all servers are responsible for exactly the same number of shards without concern for per-shard traffic.

Replication. We next add replication into the mix to combat the heavy-tailed load on shards. In a somewhat simplified summary, existing dynamic load-balancing techniques for distributed caching and storage systems operate in two phases: **(1)** *Detection*: identify hot servers and their hot contents; followed by **(2)** *Replication*: move data between servers to alleviate high load, or divide traffic across multiple servers by replicating hot content elsewhere, sometimes on many nodes. We now survey several state-of-the-art detection and replication techniques.

A ***front-end cache*** is normally a small cache deployed in front of the tier experiencing the imbalanced load [5]. The detection phase depends on the replacement algorithm used by the cache, such as LRU (Least-Recently-Used), to detect very popular objects which exhibit high temporal locality. Once the hot object is detected, its replica is stored by the front-end cache, which can then serve all traffic flowing through it, via its local copy of the item and without burden on other servers. Studies show that even small front-end caches can substantially alleviate skewed object-access workload [5, 7].

Facebook's front-end Web servers already embed a small cache for popular Tao objects. Our traces, therefore, are focused on load imbalance *after* caches higher up in the hierarchy have already been applied [7]. Moreover, our earlier analysis showed that after a layer of front-end caches, the popularities of objects are no longer a significant factor in Tao's load-imbalance. Instead, the skew partly stems from the popularity of shards that each comprise multiple correlated objects connected through the social graph. However, compared to objects, shards are just too large to be cached on front-end Web servers: typical shard size in Tao is on the order of hundreds of megabytes, while typical object size is measured in kilobytes. Therefore, further improving the front-end cache is unlikely to resolve the load-imbalance situation in Tao — other solutions are needed.

Replicating hot content. Hong and Thettethodi [6] recently proposed augmenting the cache infrastructure to actively monitor and replicate hot objects across multiple servers. In their scheme, dubbed SPORE, each `memcached` server monitors the popularity of its own content and informs clients about replication and rerouting decisions. The hotness-detection policy, in contrast to the "dominant" resource approach currently used in Tao, is implemented by maintaining a list of ranked counters, updated with an exponentially-weighted moving average for each item. Replication reconciliation is then controlled through time-based leases.

As mentioned earlier, Tao's replication component monitors shard loads on servers, and it replicates dominant shards every ten minutes. Our analysis already showed that flash crowds and surges in popularity can destabilize an unfortunate caching server within Tao's 10-minute replication window. Moreover, its reverse routine is too conservative, resulting in unnecessary memory waste.

Streaming methods. Mounting interest has been seen for *streaming algorithms*, which can process incoming data streams in a limited number of passes to provide approximate summaries of their data, including heavy-hitter identification and frequency estimation for popular items [2]. Frequency-estimation algorithms, useful for detecting hot shards, sample requests from the data stream, often at a very low rate, and carefully maintain a collection of candidates for hot shards. Frequency estimates can then be used to adaptively replicate shards based on their popularity. In an effort concurrent with ours, Hwang and Wood utilize streaming algorithms to reactively mitigate load balance [8].

Streaming algorithms feature high performance, while requiring very small memory footprints and CPU overhead. They can be parallelized through sharding, much like a cache tier for scalability. The algorithms complement other solutions, such as front-end caching, and they can be deployed transparently in an existing cache implementation.

Streaming algorithms are generally faster than cache reports at detecting hot shards, as they operate at a finer temporal granularity and can identify trends practically in real-time. Cache reports effectively serve as snapshots over larger time windows, whereas streaming algorithms maintain several summaries of shorter time intervals, thus providing a longer and more detailed access history of popular shards. This can be further leveraged to identify patterns and make predictions on upcoming access frequencies. In our streaming-algorithm implementation, the most frequently requested shards in a

Replication	Hashing		
	libketama	Tao	Perfect (Theor.)
None	1.53	1.46	1.34
Tao	1.53	1.25	1.17
Perfect (Theor.)	1.41	1.18	1.00

Table 2: *Comparison of $^{max}/_{avg}$ statistic of various hashing and replication schemes.*

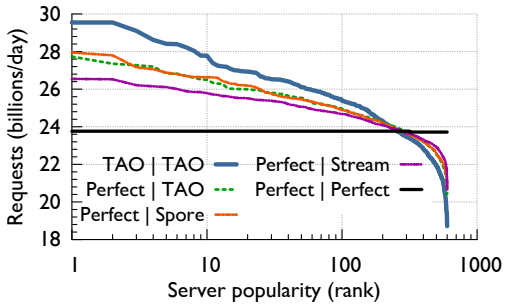

Figure 10: *Load distribution of various hashing and replication schemes (denoted in such order in figure legend).*

60-second time window are replicated to a constant number of servers. A shard is de-replicated when it has not been deemed hot within the last four minutes. With a higher sampling ratio, hot shards can be identified at much finer granularity. Moreover, traffic estimates can be used to dynamically calculate the appropriate number of replicas for a given shard.

3.3 Comparison and Evaluation

We now evaluate different hashing and replication schemes for load balancing, using our **ReqSample** trace. For each technique, we will examine the $^{max}/_{avg}$ metric, which denotes the volume of requests received on the most loaded server relative to that of an average server, across the time period (24 hours) of the full trace. Table 2 summarizes our $^{max}/_{avg}$ metric for the key hashing and replication schemes, while Figure 10 visualizes the impact of replication schemes on load imbalance in a single cluster.

When no replication mechanism is used, Tao has $^{max}/_{avg}$ of 1.46, outperforming the consistent-hashing reference implementation of `libketama` [9], with its $^{max}/_{avg}$ of 1.53. Even with theoretically perfect hashing, the best load imbalance one obtains without using any replication method is a $^{max}/_{avg}$ of 1.34, with the most loaded server 60% more burdened than the one with the lightest load. If these methods also incorporate a perfect replication scheme, the difference is even more stark: 41% more load on the most highly loaded server, than on an average one, for `libketama` compared to only 17% more for Tao. We also deduce that the hashing scheme within Tao can be improved by up to eight percentage points, from $^{max}/_{avg}$ of 1.25 to 1.17.

Retaining Tao's current hashing mechanism, there is an opportunity to improve on Tao's replication methods to decrease $^{max}/_{avg}$ from 1.25 to 1.18. Figure 10 further details how replication mechanisms affect load skew. We isolate the impact of the replication algorithm by assuming shards are uniformly distributed across servers. In this case, Tao's replication scheme achieves a $^{max}/_{avg}$ of 1.17, slightly outperforming, the lease-based replication scheme SPORE [6], which has a $^{max}/_{avg}$ of 1.18. The streaming algorithm performs significantly better at detecting hot shards than Tao's replication mechanism, but the overall reduction in load balance has room for improvement, moving $^{max}/_{avg}$ from 1.17 to 1.12. The streaming algorithm provided the most competitive replication scheme by far, but the detection and replication mechanisms can be substantially refined — part of our ongoing work.

Takeaways. (1) Standard consistent hashing results in 53% higher load on select servers relative to the average, and 240% relative to the least loaded server. Even if this hashing technique is coupled with an optimal replication algorithm, the most loaded server remains 41% more loaded than the average server. **(2)** Tao's hashing algorithm improves upon plain consistent hashing, but the most loaded server still sustains 18% more load than the average server, and 34% more than the one with the lightest load, even if the replication scheme balances per-shard load perfectly. **(3)** The use of streaming algorithms for hot-spot detection outperforms other replication schemes, but a $^{max}/_{avg}$ metric of 1.12 suggests room for future improvement.

4 Conclusion

The scalability of today's popular web sites is enabled by large clusters of in-memory cache servers. Each server in a cluster must be equipped to handle peak load, but this implies extensive overprovisioning due to load imbalance across the cache servers. We investigate the causes of the load skew on real-world traces from Facebook's Tao cluster: we identify the major roles played both by imbalanced content placement, and popularity disparity caused by dependent sharding, while we dismiss the significance of the impact from object-level popularity skewness. Through simulation, we recognize that current load-balancing techniques — including consistent hashing, and different flavors of replication — only partially address such skewness, while an approach based upon streaming-analytics holds promise for further improvement. In conclusion, our results pave the way for continued research into more effective mitigation techniques for load skew — to curb infrastructure resources and improve cache performance.

Acknowledgments

We thank our HotNets reviewers for their constructive feedback. Our work is supported, in part, by a grant from the DARPA MRC program, a grant-of-excellence (#120032011) from the Icelandic Research Fund, and additional funding from both Emory University and Facebook.

References

[1] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proc. of the 2013 USENIX Annual Technical Conference (ATC '13)*, pages 49–60, San Jose, CA, USA, 2013.

[2] G. Cormode and M. Hadjieleftheriou. Methods for Finding Frequent Items in Data Streams. *The VLDB Journal*, 19(1):3–20, February 2010.

[3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Banff, Alberta, Canada, 2001.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pages 205–220, Stevenson, WA, USA, 2007.

[5] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proc. of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*, pages 23:1–23:12, Cascais, Portugal, 2011.

[6] Y.-J. Hong and M. Thottethodi. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Proc. of the 4th ACM Symposium on Cloud Computing (SOCC '13)*, pages 13:1–13:17, Santa Clara, CA, USA, 2013.

[7] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proc. of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 167–181, Farminton, PA, USA, 2013.

[8] J. Hwang and T. Wood. Adaptive Performance-Aware Distributed Memory Caching. In *Proc. of the 10th International Conference on Autonomic Computing (ICAC '13)*, pages 33–43, San Jose, CA, USA, 2013.

[9] R. James. `libketama`: a consistent hashing algo for memcache clients. `http://github.com/RJ/ketama` (accessed on 2014/07/15), April 2007.

[10] 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 *Proc. of the 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, pages 654–663, El Paso, TX, USA, 1997.

[11] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web Caching with Consistent Hashing. In *Proc. of the 8th International World Wide Web Conference (WWW '99)*, pages 1203–1213, Toronto, Ontario, Canada, 1999.

[12] A. Likhtarov, R. Nishtala, R. McElroy, H. Fugal, A. Grynenko, and V. Venkataramani. Introducing mcrouter: A memcached protocol router for scaling memcached deployments. `http://tinyurl.com/n5t338j`, September 2014.

[13] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*, pages 385–398, Lombard, IL, USA, 2013.

[14] X. Wang and D. Loguinov. Load-balancing Performance of Consistent Hashing: Asymptotic Analysis of Random Node Join. *IEEE/ACM Transactions on Networking*, 15 (4):892–905, August 2007.

[15] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving Cash by Using Less Cache. In *Proc. of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '12)*, Boston, MA, USA, 2012.