The Case for Dynamic Cache Partitioning for Tail Latency

Daniel S. Berger*[1], Benjamin Berg[2], Timothy Zhu[2], and Mor Harchol-Balter[3]
[1]*Student at University of Kaiserslautern* [2]*Student at Carnegie Mellon University* [3]*Carnegie Mellon University*

The tail latency challenge. Large webservices like Amazon and Google frequently emphasize the role of low tail latency (e.g., the 99 percentile of the latency) in user satisfaction [7]. Unfortunately, maintaining low tail latencies in real-world systems is a challenging problem [1, 7, 8, 13].

How real-world traffic affects tail latency. As a case study, we consider tail latency at Wikipedia.org, one of the world's busiest websites. A simplified view of Wikipedia's architecture includes two backend clusters as shown in Figure 1a. The "text" cluster is a distributed database which contains the text of all Wikipedia articles. The "upload" cluster is a media storage system for photos and other media shown in Wikipedia articles. Figure 1b shows the overall rate of the two types of requests during the 2016 US election week. We find that over five days, the daily peak rate increases by 33% for text requests, while remaining mostly constant for upload requests. The increased text cluster load leads to significantly increased tail latency for text requests. As Wikipedia articles consist of both text and upload content, user latency is determined by the maximum of the two latencies.

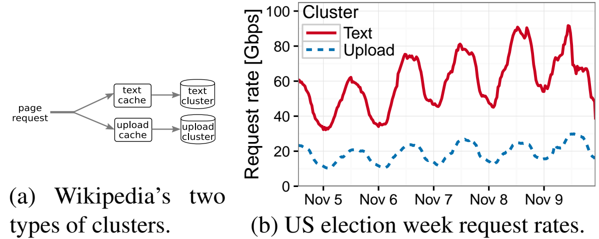
Goal: minimize maximum tail latency. Formally, if $P99_{text}$ and $P99_{upload}$ denote the 99-percentile latency for text and upload requests, respectively, our goal is to minimize the maximum tail latency:

$$\min \left\{ \max \left\{ P99_{text}, P99_{upload} \right\} \right\} . \quad (1)$$

Why classical load-balancing is not applicable. One might think of shifting load from the text cluster to the upload cluster. Unfortunately, this does not work because the two clusters require different software and data.

A novel approach to solving the load imbalance problem. Instead of using separate dedicated text and upload caches, as is currently done at Wikipedia, this poster proposes using a shared cache for both types of requests. As caches serve as a filter for reducing requests sent to backend systems, a shared cache would enable us to cache more text requests, and thus reduce load on the text cluster. Unfortunately, there are several challenges in building such a shared caching system.

Why classic shared caching policies fail. The simplest configuration shares the capacity between text and upload, and uses LRU eviction. Unfortunately, LRU does not work well for sharing due to its inherent bias against infrequently accessed objects. For example, the lower request rate to objects in the upload cluster results in a 10x higher P99 latency than the P99 latency of text requests, as shown in Figure 2a.



(a) Wikipedia's two types of clusters. (b) US election week request rates.

Figure 1: (a) Simplified architecture of Wikipedia.org: there are two large backend systems (the "text" and the "upload" cluster), each with a dedicated caching system with statically allocated resources. (b) This static setup leads to different loads for the backend systems as their request rates behave differently over time.

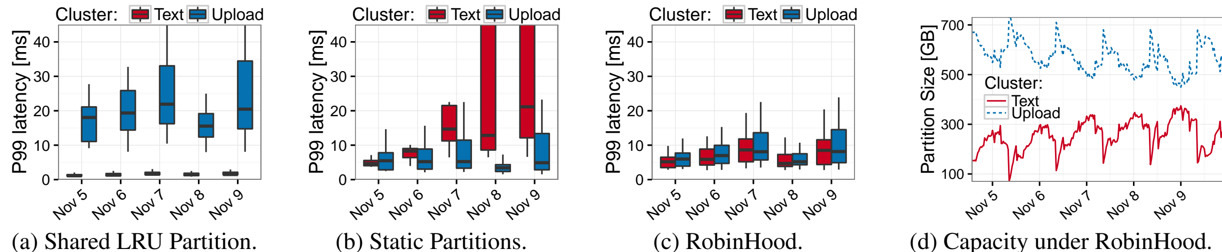
While there are many more sophisticated caching policies than LRU [2, 4], they either ignore latency, or prove to be ineffective for tail latency in our experiments.

Why static cache partitions fail. To mitigate LRU's bias, some caching systems statically partition the cache space for each request type [3, 12]. We experimentally determine the optimal partition for November 5 and the week prior (which is similar). Figure 2b shows the corresponding P99 latency: while text and upload latencies are low before November 6, text latency is 3-5x higher on average during November 7-9.

Why prior dynamic partitioning techniques fail. Instead of a static partitioning, several prior works have proposed dynamic partitioning algorithms. These algorithms focus on maximizing the overall hit ratio [5, 6, 9, 10]. Unfortunately, hit ratio is not a good metric for representing the tail latency performance of a partition. A similar problem occurs with fair-allocation algorithms [11].

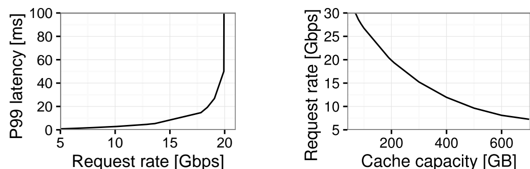
RobinHood: tail-latency-aware dynamic partitioning. We propose a new partitioning algorithm, called RobinHood, which minimizes Eq.(1). Our proposal relies on two observations. Firstly, tail latency occurs mostly due to excessive queueing in the system [8, 12], and queueing theory tells us that the relationship between request-rate and P99-latency is convex, as shown in Figure 3a.

*Corresponding author: berger@cs.uni-kl.de



(a) Shared LRU Partition. (b) Static Partitions. (c) RobinHood. (d) Capacity under RobinHood.

Figure 2: We simulate a simplified and scaled-down version of Wikipedia.org on the trace from Figure 1. The total cache capacity is 1TB, and there always exists a cache capacity allocation that keeps the cluster load below 40%. We track the P99 latency over 30min intervals for text requests and upload requests and show aggregate box plots for each day. We find that (a), a shared LRU partition results in high latency for upload requests, (b), static partitions result in high latency for text requests, and (c), our novel RobinHood algorithm achieves low latency for both request types on all days. (d) RobinHood dynamically partitions cache capacity between text and upload requests over time.



(a) Request rate vs P99 latency. (b) Capacity vs. request rate.

Figure 3: Simulation results of the text cluster with traces from November 8. (a) The P99 (99-percentile) latency is a convex function of the cluster's request rate. (b) The cluster's request rate is a convex function of the allocated cache capacity. The upload cluster behaves similarly.

Secondly, there are diminishing returns from additional cache capacity, which leads to a convex relationship between cache capacity and backend request-rate[1], as shown in Figure 3b. Our two observations lead to a convex relationship between cache capacity and P99 latency. Therefore, a simple hill climbing algorithm will be theoretically optimal for minimizing the P99 latency. Specifically, RobinHood measures the P99 latency per partition over intervals of 100K requests. At the end of the interval, RobinHood transfers 1% of the capacity of the partition with the lowest P99 latency to the partition with the highest P99 latency.

Improving tail latency via RobinHood. Figure 2c shows that RobinHood keeps the P99 latencies for text and upload requests low and within 10% of each other on every day. Compared to Static Partitioning, RobinHood improves the tail latency of text by 40-60% on November 7-9. Compared to Shared LRU, RobinHood improves the tail latency of upload requests by 60% on average.

Figure 2d shows how RobinHood partitions the capacity over time. RobinHood makes more than 500 changes, and prevents high load to the text cluster by allocating an increasing fraction of cache capacity for text requests.

Conclusions. This poster shows that dynamic partitioning of cache space can be an effective method for tail-latency-aware load balancing accross different backend systems. As many webservices use cache partitions [12], we believe that this type of load balancing can prove useful in many other contexts besides Wikipedia.org.

[1]This is true for highly multiplexed traffic, e.g., Wikipedia.org. In other cases, the method from [6] can convexify the curve.

References

[1] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI*, pages 19–19, 2012.

[2] D. S. Berger, R. Sitaraman, and M. Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a CDN. In *USENIX NSDI*, 2017. To appear.

[3] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, pages 49–60, 2013.

[4] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *USENIX SITS*, pages 193–206, 1997.

[5] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: dynamic cloud caching. In *USENIX HotCloud*, 2015.

[6] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*, 2016.

[7] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, 2013.

[8] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! In *USENIX NSDI*, pages 9–21, 2015.

[9] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *USENIX ATC*, pages 57–69, 2015.

[10] 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.

[11] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. Fairride: Near-optimal, fair cache sharing. In *USENIX NSDI*, pages 393–406, 2016.

[12] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song. Kraken: leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *USENIX OSDI*, pages 635–650, 2016.

[13] T. Zhu, D. S. Berger, and M. Harchol-Balter. SNC-Meister: Admitting more tenants with tail latency SLOs. In *ACM SoCC*, pages 374–387, 2016.