Dynamically Sharing Memory between Memcached Tenants using Tingo

AmirHossein Seyri
University of Illinois at Chicago
aseyri2@uic.edu

Abhisek Pan
Microsoft
abpan@microsoft.com

Balajee Vamanan
University of Illinois at Chicago
bvamanan@uic.edu

ABSTRACT

Web applications utilize in-memory caching systems to reduce the load on backend databases and improve the performance of the system. These cache environments normally host multiple tenants simultaneously, signifying the need to efficiently manage the underlying physical memory allocation in these environments. Off-the-shelf caches statically divide the memory between tenants, which often leads to poor utilization and low hit rates for some of these tenants. In this work, we present *Tingo*, a multi-tenant cache environment designed to adequately manage the memory allocation of cache tenants to help them adapt to their workloads and optimize their hit ratios, by dynamically reallocating memory pages among them.

CCS CONCEPTS

• **Computer systems organization** → *Distributed architectures*;

KEYWORDS

Multi-tenancy, In-memory Key-value Stores, Shared-memory, Memcached

ACM Reference Format:
AmirHossein Seyri, Abhisek Pan, and Balajee Vamanan. 2019. Dynamically Sharing Memory between Memcached Tenants using Tingo. In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19 Companion), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3360468.3366782

1 INTRODUCTION

In-memory key-value stores are a critical part of web services. In-memory caching systems such as Memcached [7] are a popular solution widely used by major web service providers such as Facebook to reduce the load on backend databases and the web request latency [1]. Memcached is an in-memory cache that receives the data by a key-value API and stores them on DRAM using a hash table. It uses *slabs* to store objects of varying sizes and prevent memory fragmentation. In slab allocation, the total memory allocated to the cache is divided into small and fixed-size pieces. Objects (items) are then categorized into slab classes depending on their size, and those with similar sizes will be stored in the same class. When the cache is empty, upon receiving new items, a new memory page (1MB) is allocated to the cache and assigned to this particular slab class. Each class has a separate queue to keep track of the order of items and runs LRU (Least Recently Used) as the eviction policy.

Major cloud providers such as Microsoft offer in-memory key-value storage [6] as a service to thousands of customers. However, multiple tenants[1] must share underlying resources such as CPU, memory, and I/O among them. Because of the deployment scale of in-memory caches and their ability to cut latency by multiple orders of magnitude compared to disks, minor improvements in memory utilization and hit rate of them will lead to a significant increase in performance of web applications [3] and will reduce their operating costs. For example, consider a web application utilizing a Memcached instance with a 97% hit ratio. If the average cache latency was 100 μs and the access latency of the database was 10 ms, then the total expected latency of the application will be 397μs (= 0.97 × 100μs + 0.03 × 10ms). If the cache hit ratio gets to increase by only 1%, the total latency will be reduced to 298μs (= 0.98 × 100μs + 0.02 × 10ms), which is a 25% improvement.

The most common approach of sharing the cache is by statically partitioning the main memory among these tenants. This technique is inefficient since most workloads are dynamic and differ in aspects such as working set size, access pattern, and the relationship between hit rates and cache sizes (i.e., hit rate curves). There might be a tenant needing more space while another tenant is not fully utilizing its cache capacity. Because of the possibility of rapid changes in the workloads, in order to keep the overall throughput of the system high, memory allocation of the tenants should be adjusted frequently and manually, making this approach expensive.

Recent work such as Cliffhanger [2] greedily optimizes the space, but only *within* the tenant. Cliffhanger uses an algorithm that runs in addition to the Memcached system and optimizes the memory allocation by computing the gradients of hit rate curve for each eviction queue. It is able to reallocate space from queues that least benefit from memory to queues that benefit the most. Memshare [3] proposed a system by utilizing the *Cliffhanger* algorithm in a cache that uses a *log-structured memory* instead of a slabbed memory. We make the key insight that an ideal multi-tenant cache environment should adapt to the working set size of tenants by reallocating memory from tenants who do not need it at the moment, and giving it to tenants that benefit the most.

In this work, we propose *Tingo*[2], the first greedy algorithm that optimizes the hit rate of all tenants while minimizing the impact on fairness among tenants. Cliffhanger is limited to the slabs *within* the tenant, whereas in our system, the slabs of other tenants are also considered as a source of memory. Our proposal is based on using

ACM ISBN 978-1-4503-7006-6/19/12…$15.00
https://doi.org/10.1145/3360468.3366782

[1]We use the words "tenants" and "applications" interchangeably throughout the paper.
[2]"Tingo" is a word from Easter Island, and it means to borrow objects from a friend's home one-by-one until there is nothing left.[5]

shared-memory for multiple Memcached instances. Shared-memory allows the system to efficiently move memory pages between these processes.

2 TINGO

In Tingo, a segment of memory is shared between different Memcached tenants. This shared segment is initialized by setting up its size. Every tenant requesting a page will be allocated space inside the segment. A small piece of memory called *Tracker* will also be shared between tenants and used as a central entity controlling the use of the shared-memory and managing the allocation and reallocation of the memory pages. The memory could be initially allocated to the tenants either *statically*, that is the same as the traditional cache sharing mechanism in which the system is not able to reallocate the memory, or *dynamically (greedy)*, in which there is no memory limit for any of the Memcached instances and each one can use the shared-memory as much as they have data to store.

Tingo has the ability to move the memory pages by utilizing *shadow queues*, similar to Cliffhanger [2]. However, unlike Cliffhanger, Tingo moves memory, not only between slab classes within a tenant, but also across tenants. Thus, Tingo adapts to the changes in their workloads. Every queue of each slab class will have a shadow queue as an extension that is used to keep the most recent evicted items. By counting the hits on the shadow queue, it's able to estimate the hit rate on a virtually extended main queue. The queue with higher hits would benefit more from extra memory and will be granted a new page, at the expense of another queue losing one. The important feature the distinguishes Tingo from previous work is that, they are only able to resize a slab class by resizing another slab class from the same tenant, whereas Tingo considers *all* slab classes across *all* tenants.

This provides Tingo with greater *flexibility* for memory allocation, in terms of both *total allocated size* and internal *queue sizes* (memory allocated to a each slab class), meaning Tingo tenants are not bound to their initial allocated memory limit. In other words, they can grow their total size if they need to or resize the space allocated to each slab class (queue sizes) according to the workload they are receiving, or even give away their spare memory pages to another tenant and reduce their size.

The shared-memory is carefully managed and distributed among tenants, in such a way that no tenant is able to access the memory pages allocated to other tenants. When a page is chosen to move between tenants, all items stored in that page will be evicted from the tenant's queue, and the memory is fully cleared before getting reallocated, preventing the new tenant from accessing the previous owner's items.

3 SYSTEM EVALUATION

We implemented Tingo on top of Memcached and evaluated it using Cloudsuite [4] benchmark tool and a pre-built Twitter dataset. We scaled the dataset to 10 different sizes (1 to 10GB), and ran 14 sets of benchmarks for 4 tenants (total 56), each with a different workload size. In all benchmarks, the shared-memory was 16GB in size and equally divided among tenants (4GB each), which was equal to the total size of workloads of the 4 tenants participating in the set.

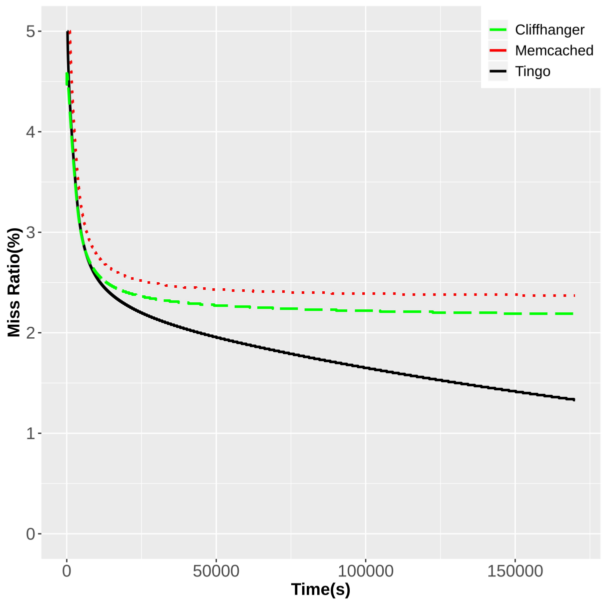

Figure 1: Miss Ratio curves for a demanding tenant, comparing the 3 models (Memcached, Cliffhanger and Tingo). Initial allocated memory is 4GB and the workload size is 8GB.

We measured the hit ratio improvements and memory allocations of these 56 tenants over a 48-hour duration. In addition to *Tingo*, we evaluated the base model of Memcached and base Cliffhanger using the same settings and physical testbed and compared them to Tingo. We ran these experiments on two machines: a 64-core 2.00GHz Intel(R) Xeon(R) CPU E7-4820, and a 64-core 2.4GHz AMD Opteron(tm) Processor 6378, both with 256GB of DDR3 DRAM, running Ubuntu Server 18.04 and Linux kernel version 4.15.0-58.

3.1 Results

The evaluation results show that, compared to Cliffhanger, Tingo decreases the miss rates by an average of 49% and a maximum of 73% for the demanding tenants (tenants that are allocated smaller space compared to their workload size). Figure 1 demonstrates the miss ratio curves for one of these tenants that was initially allocated memory half of its workload size. The memory size of this tenant was initially 4GB, and its final allocated space after the duration of benchmarks was 5.2GB.

Our preliminary algorithm does not take *fairness* into consideration, but we plan to consider fairness in our future work.

REFERENCES

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.
[2] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 379–392.
[3] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. 2017. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 321–334.
[4] Cloudsuite. 2019. Cloudsuite - A Benchmark Suite for Cloud Services. (2019). http://cloudsuite.ch/ [Online; accessed 20-September-2019].

[5] A.J. de Boinod. 2006. *The Meaning of Tingo: And Other Extraordinary Words from Around the World*. Penguin Adult. https://books.google.com/books?id=pzrxvAFZsuEC

[6] Redis Labs. 2019. Redis. (2019). https://redislabs.com/ [Online; accessed 20-September-2019].

[7] Memcached. 2019. memcached - a distribtued memory object caching system. (2019). https://memcached.org [Online; accessed 20-September-2019].