Modeling and Analyzing Latency in the Memcached system

Wenxue Cheng[1], Fengyuan Ren[1], Wanchun Jiang[2], Tong Zhang[1]
[1]Tsinghua National Laboratory for Information Science and Technology, Beijing, China
[1]Department of Computer Science and Technology, Tsinghua University, Beijing, China
[2]School of Information Science and Engineering, Central South University, Changsha, China

March 27, 2017

abstract

Memcached is a widely used in-memory caching solution in large-scale searching scenarios. The most pivotal performance metric in Memcached is latency, which is affected by various factors including the workload pattern, the service rate, the unbalanced load distribution and the cache miss ratio. To quantitate the impact of each factor on latency, we establish a theoretical model for the Memcached system. Specially, we formulate the unbalanced load distribution among Memcached servers by a set of probabilities, capture the burst and concurrent key arrivals at Memcached servers in form of batching blocks, and add a cache miss processing stage. Based on this model, algebraic derivations are conducted to estimate latency in Memcached. The latency estimation is validated by intensive experiments. Moreover, we obtain a quantitative understanding of how much improvement of latency performance can be achieved by optimizing each factor and provide several useful recommendations to optimal latency in Memcached.

Keywords
Memcached, Latency, Modeling, Quantitative Analysis

1 Introduction

Memcached [1] has been adopted in many large-scale websites, including Facebook, LiveJournal, Wikipedia, Flickr, Twitter and Youtube. In Memcached, a web request will generate hundreds of Memcached keys that will be further processed in the memory of parallel Memcached servers. With this parallel in-memory processing method, Memcached can extensively speed up and scale up searching applications [2].

Latency is the most pivotal performance metric of the Memcached [2]. Previous works have verified that the latency is affected by various factors, including the workload pattern [3], the service rate [4, 5], the unbalanced load distribution [6] and the cache miss ratio [7]. The workload pattern refers to both the average rate and the burst degree of key arrivals at each Memcached server, and the load distribution presents how loads are distributed among different Memcached servers. Focused on each factor, many proposals have been developed to improve the latency performance in Memcached systems, such as reducing the load for each Memcached server [2], utilizing high performance interconnection technology such as InfiniBand [8, 9] and RDMA [10, 11] to improve service rate, optimizing the replication algorithms to improve the imbalance among Memcached clusters [12, 13], and reducing cache miss ratio [7]. However, these proposals only focus on some partial factors, and there is still no comprehensive and quantitative understanding of which factor has the most significant impact on the latency and how much improvement on latency can be achieved by optimizing each factor. Although recent analytical work [14, 15] has presented some latency estimations for a general distributed service system, they own the following three limitations to apply to Memcached scenarios. 1) Does not address unbalanced load distribution among Memcached servers. 2) Apply to independent Poisson traffic but cannot handle the burst and concurrent key

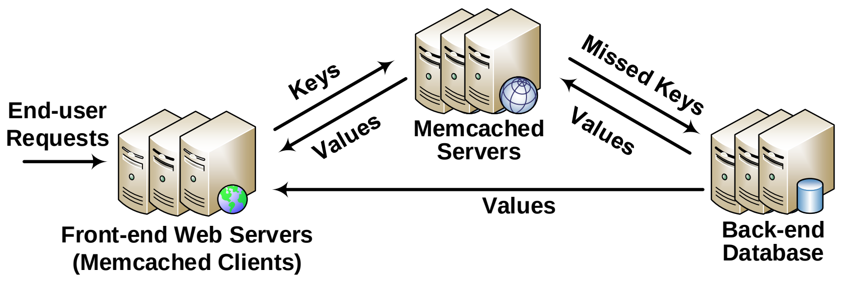


Figure 1: Memcached Architecture.

arrivals at each Memcached server. 3) Cannot accurately describe the two-stage processing of an end-user request in the Memcached system.

This paper aims to quantitate the impact of each factor on Memcached latency. To address the limitations of previous models, we establish a specific model for the Memcached system through three measures. 1) Introducing a set of probabilities to formulate the unbalanced load distribution among Memcached servers. 2) Developing a $GI^X/M/1$ model to formulate the queueing process at Memcached servers, where the burst and concurrent keys are regarded as batching blocks. 3) Adding a cache miss processing stage with $M/M/1$ queues. Based on this model, algebraic derivations are conducted to estimate the latency in Memcached, which is separated into three parts, i.e. the network latency, the processing latency at Memcached servers, and the processing latency at database. The latency estimation is validated under various realistic configurations. Moreover, we conduct a quantitative analysis on the latency estimation to demonstrate which factor has the most significant impact on the latency of Memcached and how much improvement can be achieved by optimizing each factor. Insights and recommendations via modeling and analyzing the latency in Memcached are are twofold.

1) The processing latency at Memcached Servers reaches a cliff point when the server utilization gets to a specific value, which is negatively related with the burst degree of keys. Under the Facebook workload, the specific utilization is about 75%. To ensure low latency, we recommend that the Memcached server utilization should keep below the specific utilization and load-balancing mechanisms should take effect before and Memcached server extends the specific utilization.

2) The latency of end-users grows logarithmically as an end-user request generates more Memcached keys or the cache miss ratio increases. Since an end-user request always generates numerous Memcached keys and the cache miss ratio is always quite tiny, we recommend drastically decreasing the number of Memcached keys generated from an end-user request instead of reducing the tiny cache miss ratio.

The rest of the paper is organized as follows. Section 2 presents an overview of the Memcached architecture as well as the related work. Subsequently, a specific model for Memcached is established in Section 3 and the latency estimation is deduced in Section 4. Further in Section 5, we conduct intensive experiments to validate the latency estimation, quantitate the impact of each factor on latency in Memcached, and summary the key insights and recommendations of improving latency performance in the Memcached system. Finally, the paper is concluded in Section 6.

2 Background and Related Work

2.1 Overview of Memcached

During the past 10 years, Memcached has been widely deployed in many large-scale websites such as Facebook, LiveJournal, Wikipedia, Flickr, Twitter and Youtube. By caching database results in the high-speed memory, Memcached avoids reading data from the low-speed disk, thus improves the user experience.

As illustrated in *Fig.*1, in a typical Memcached system, the Memcached clients are deployed in the front-end web servers, and the Memcached servers are located between the front-end web servers and the back-end database. Data are stored as key-value items. When an end-user request arrives, the Memcached client transforms the request into hundreds of keys. For each key, one Memcached server is selected based on a key-to-server hashing algorithm for further processing. If the corresponding value has been cached, the Memcached server returns it. Otherwise, the missed key will be relayed to the back-end database. Till all the values for these keys are returned, the original end-user request is completed.

In addition, traffic of Memcached has some specific features.

1) The arrival of keys is concurrent and burst. According to the measurement results in Facebook [3], two or more keys might arrive at one Memcached server during a tiny time($< 1\mu s$) with a large probability of 0.1159, and the inter-arrival gap of keys follows a Generalized Pareto distribution, which is heavy-tailed and indicates burst arrival of keys.

2) The load distribution among different Memcached servers is unbalanced. Based on the statistical results in Facebook [3], a small percentage of values are accessed quite frequently, while the rest numerous ones are accessed only a handful of times. Such that Memcached servers caching the popular items have to handle a heavy load, while other Memcached servers are light-loaded.

2.2 Optimizations of Latency in Memcached

Latency is the most pivotal performance metric in a Memcached system. There is a lot of work attempting to optimize latency in realistic Memcached deployments. We find that the optimization work mostly focuses on the following factors.

1) **Load on one Memcached server.** Heavy load for one Memcached server always results in poor latency of processing. To improve latency, systems like Facebook [2] always deploy numerous Memcached servers in their Memcached clusters such that the average load size for one Memcached server is appropriate .

2) **Service rate of Memcached servers.** The latency performance in Memcached is always limited by the service rate of Memcached servers. Especially when the 10 Gigabit Ethernet has been widely used in the data centers, the bottleneck limiting latency reduction lies not in the network but in Memcached servers. In Facebook, keys may arrive at one Memcached server at a maximum speed of 10^5 per second, and the average inter-arrival gap is $16\mu s$ [3]. While a 10 Gbps link can transmit at most 6×10^6 keys (no larger than 200B) or 1.2×10^6 values (no larger than 1KB) per second, the average cost of Memcached servers to handle a single key can even be up to $12\mu s$ [16]. Consequently, the network utilization is less than 10% while the Memcached server utilization is as high as 75%. Thus, improving the service rate of Memcached servers can significantly reduce the latency. In this context, many relevant optimizations are proposed. Intel overcomes the thread-scaling limitations of Memcached servers [17], and provides multi-core configuration guidelines [18]. High performance technologies such as InfiniBand [8, 9] and RDMA [10, 11] have been adopted to improve the performance of Memcached. Besides, the Memcached architecture has already been implemented on FPGAs [4] and embedded cores [5] to reduce latency.

3) **Unbalanced load distribution among Memcached servers.** If some Memcached servers handle a heavy load while others are light-loaded, the processing latency in heavy-loaded servers is much larger than that in light-loaded servers. However, latency of an end-user request is always determined by the maximum latency of the keys generated by it. Thus, the unbalanced load distribution among Memcached servers has a negative impact on latency. To overcome this problem, load balance mechanisms, such as improving the replication algorithms [12, 13], are introduced in Memcached.

4) **Cache miss ratio.** It has been convinced that even modest improvement of the cache miss ratio impacts the latency performance a lot for Memcached [7]. For example, assuming the average read latency from the cache and the database are $200\mu s$ and $10ms$, respectively, reducing the cache miss ratio from 2 % to 1 % would reduce the average latency from $400\mu s$ to $300\mu s$. Several systems learn from the newest cache technologies and attempt to reducing the cache miss ratio to improve the latency. GD-Wheel (GDW) [19] and GD-SizeFrequency [20] modify their cache allocation and eviction policies; Cliffhanger [7], Dynacache [21], Moirai [22], Mimir [23] and Blaze [24] optimize their resource allocation based on the hit rate curve; Twitter and Facebook [2] improve their Memcached slab class allocation to better adapt to varying item sizes.

Besides above factors, **the number of Memcached keys generated from one end-user request** is another factor impacting the latency in Memcached. Since fetching more items results in waiting for longer time, many websites, such as Google and Wikipedia, have tried to minimize the number of items in their pages to improve the latency performance.

To sum up, there are diverse factors that impact latency in Memcached and optimizing any of these factors seems to be effective. However, it is difficult to judge which one factor has the most significant impact on the latency and how much improvement on latency can be achieved by optimizing each factor. As far as we are concerned, few previous theoretical models are specific to latency in Memcached. In this paper, we will show a

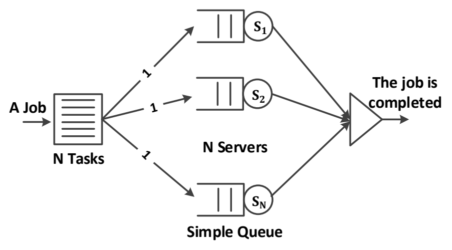


Figure 2: A typical Fork-Join model

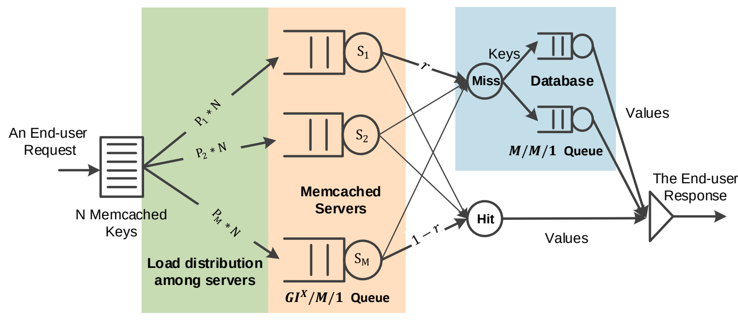


Figure 3: Specific model for Memcached.

quantitative understanding of the latency in Memcached through a modeling and analyzing method.

2.3 Fork-Join Model

The Fork-Join model has been widely used to depict the behavior of parallel processing in many data analytical systems like MapReduce [25] and Amazon's EC2 [26]. It captures similar features of Memcached. As illustrated in *Fig.*2, in a typical Fork-Join model, a job is split into N tasks to be parallelly processed by N servers and the job will be completed after all its tasks are serviced. Based on this model, latency estimations [14, 27–30] have been proposed for a general distributed service system. However, the typical Fork-Join model and the latency estimations can not be directly used for Memcached, due to the following limitations.

1) One-to-one task distribution. A typical Fork-Join model assumes that a job arriving at the system consists of the same number of tasks as the number of servers [27–29]. That is, each server only serves one task for a job. But in Memcached, keys generated from an end-user request are distributed among all available Memcached servers based on a key-to-server hash algorithm. In this case, some Memcached servers need to process more than one keys for an end-user request. Even worse, the load distribution among Memcached servers is always unbalanced, and this unbalanced load distribution has a rather negative impact on latency. Therefore, the one-to-one task distribution in the typical Fork-Join model is not matched with Memcached systems.

2) Simple service queues. Although the typical Fork-Join model has been extended to multiple service queues [14, 30], the concurrent arrivals at servers has not been taken into consideration. However, in Memcached, keys always arrive burstily and concurrently at each Memcached server, and these burst and concurrent keys are likely to introduce large latency. Therefore, the typical Fork-Join model based on simple service queues fails to quantitate the negative impact of burst and concurrent keys on latency in Memcached.

3) One single processing stage. In the typical Fork-Join model, a job is immediately completed after each task is serviced by its corresponding server. However, in Memcached, the processing results of keys at Memcached servers might be missed, and the missed keys will be relayed to the back-end database. Although the cache missing ratio is rarely tiny, the huge latency of reading data from the database makes the missing cases non-ignorable. Thus, the typical Fork-Join model that has only one single processing stage can not capture such cases of Memcached to analyze the latency.

3 Specific Model for Memcached

This section presents our specific model for the Memcached system. As illustrated in *Fig.*3, addressing the limitations of the typical Fork-Join model, we develop our own model that applies to Memcached systems. Relative to the typical Fork-Join model, the new model mainly includes the following three enhancements.

1) We introduce a set of probabilities $\{p_j\}_{j=1}^{M}$ to formulate the unbalanced load distribution among Memcached servers. On one hand, $\sum_{j=1}^{M}\{p_j\} = 1$ and each probability $\{p_j\}$ denotes that on average $p_j * N$ out of N keys generated from an end-user request are hashed to Memcached server S_j. On the other hand, the probabilities $\{p_j\}_{j=1}^{M}$ indicate the average load of all Memcached servers follow the proportional relation $\{p_1, p_2, \cdots, p_M\}$. When $p_j \not\equiv \frac{1}{M}$, the load distribution among Memcached servers is unbalanced.

2) Considering the burst and concurrent key arrivals as batching blocks, we develop a $GI^X/M/1$ model to formulate the queueing process at Memcached servers. Specially, the $GI^X/M/1$ queue has three implications.

i) GI implies that the arrival of keys can be not only Poisson but also any other pattern. For instance, the inter-arrival time of keys in Facebook follows a Generalized Pareto distribution [3], which is heavy-tailed and indicates bursts. We just assume the key arrivals at each Memcached server are independent in this model. Actually, the keys are generated by independent end-user requests from numerous Memcached clients. At each Memcached server, keys from different end-user requests are independent, and the number of keys belonging to the same end-user request is quite limited relative to the number of simultaneous end-user requests. Thus, the assumption of independent key arrivals is acceptable.

ii) X means that the concurrent arrivals are formulated as batching blocks. According to the measurement results in Facebook [3], two or more keys might arrive during a tiny time ($< 1\mu s$) with a large probability of 0.1159. Let q denote the concurrent probability. Since all the keys that arrive at the same Memcached server are independent, the batch size X, i.e. the number of concurrent keys, follows a *Geometric distribution*,

$$P\{X = n\} = q^{n-1}(1-q)$$

iii) M shows that the service time of each key follows an exponential distribution, which is a closed form for the response time CDF in practical key-value systems according to [12]. Moreover, similar assumption for service time is also adopted in recent work for Memcached, such as Chronos [31] and C3 [13].

3) We add a cache miss processing stage, where the service pattern are formulated as a $M/M/1$ queuing model. Memcached servers have a cache miss ratio r, with which probability the missed keys will be relayed to the back-end database. The cache miss processing stage is formulated as a $M/M/1$ queuing model for three reasons.

i) The unbalanced load distribution among database servers is ignored, because the database is greatly offloaded in Memcached [3, 17, 18] and the variation of load size among database servers becomes negligible.

ii) The arrival of missed keys at database servers approximately follows a Poisson process, because the missed keys can be regarded as randomly selected from the departure processes of Memcached servers, where the service time is exponential.

ii) The service time of each key in the database servers follows an exponential distribution, similar with the service time at Memcached servers discussed above.

In summary, taking all of *unbalanced load distribution among Memcached servers*, *burst and concurrent key arrivals at Memcached servers* and *cache miss processing stage* into account, our model breaks the limitations of the typical Fork-Join model, and matches the Memcached system exactly.

4 Latency Estimation

Based on our established model, this section presents the algebraic derivations of the latency estimation in Memcached. Henceforth, we use $T(t)$ and $(T)_k$ to denote the *cumulative probability function* (CDF) and the k^{th} quantile of a stochastic time T, respectively. Key notations are summarized in Table 1 for the sake of terseness.

4.1 Latency Composition

Let $T(N)$ denote the latency of an end-user request which generates N Memcached keys, and we index the keys from 1 to N. For the i_{th} key, the latency T_i between its generation and fetching the corresponding value mainly consists of three parts,

$$T_i = n_i + s_i + d_i$$

where n_i is the network latency, s_i is the processing latency at Memcached servers, and d_i is the processing latency at database. The end-user latency $T(N)$ is determined by the maximum latency of the N keys,

$$T(N) = \max_{i=1,\cdots,N}\{T_i\} = \max_{i=1,\cdots,N}\{n_i + s_i + d_i\}$$

Table 1: Key Notations

Notation	Definition
N	Number of Memcached keys generated form an end-user request.
$T(N)$	Latency of an end-user request which generates N Memcached keys.
$T_N(N)$	Maximum network latency of N keys.
$T_S(N)$	Maximum processing latency of N keys at Memcached servers.
$T_D(N)$	Maximum processing latency of N keys at database.
M	Number of Memcached servers.
$\{p_j\}_{j=1}^{M}$	Load distribution among Memcached servers.
q	Concurrent probability of keys for each Memcached server.
X	Batch size of the concurrent keys.
$\mathcal{T}_X$	Inter-arrival gap of batched keys for one Memcached server.
λ	Average rate of key arrivals, $\lambda = E[X]/E[\mathcal{T}_X]$.
μ_S	The average service rate at Memcached servers.
δ	Unique root of $\delta = \mathcal{L}_{\mathcal{T}_X}((1-\delta)(1-q)\mu_S)$ in $(0,1)$.
r	Cache miss ratio of Memcached servers.
μ_D	The average service rate at database.

Obviously, $T(N)$ is bounded by

$$\max\{T_N(N), T_S(N), T_D(N)\} \le T(N) \le T_N(N) + T_S(N) + T_D(N) \tag{1}$$

where

$$\begin{cases} T_N(N) \triangleq \max\limits_{i=1,\cdots,N}\{n_i\} \\ T_S(N) \triangleq \max\limits_{i=1,\cdots,N}\{s_i\} \\ T_D(N) \triangleq \max\limits_{i=1,\cdots,N}\{d_i\} \end{cases}$$

That is, the end-user latency $T(N)$ can be estimated by the maximum network latency $T_N(N)$, the maximum processing latency at Memcached servers $T_S(N)$, and the maximum processing latency at database $T_D(N)$. Subsequently, we will discuss these three latency in detail.

4.2 Network Latency

In this part, we will derive the maximum network latency, i.e. $T_N(N) = \max\limits_{i=1,\cdots,N}\{n_i\}$.

For the i_{th} key, the network latency n_i consists of the constant propagation and transmission delay, as well as the dynamic queueing delay. In Facebook, the aggregate workload arriving at one server can reach at most 10^5 requests per second [3], while a $10Gbps$ link can transmit more than 6×10^6 "Keys" (no larger than $200B$) requests or 1.2×10^6 "Values" (no larger than $1KB$) per second, which implies that the network utilization is no more than 10%. Therefore, there are almost no queueing in the network thus the queueing delay can be ignored. Consequently, the network latency n_i can be considered to be constant.

Therefore, the Maximum network latency of N keys is also constant,

$$T_N(N) = \max_{i=1,\cdots,N}\{n_i\} = constant \tag{2}$$

4.3 Processing Latency at Memcached Servers

In this part, we will estimate the maximum processing latency at Memcached servers, i.e. $T_S(N) = \max\limits_{i=1,\cdots,N}\{s_i\}$. We will obtain the processing latency of one key at Memcached servers from the $GI^X/M/1$ queuing model first, and then deduce the maximum processing latency of N keys.

4.3.1 The processing latency of one key

According to our model (Section 3), the queueing process at Memcached servers is formulated as a $GI^X/M/1$ queuing model. In this model, the inter-arrival gap of batched keys $\mathcal{T}_X$ follows a general distribution of $\mathcal{T}_X(t)$, the batch size X follows a geometric distribution with a mean of $\frac{1}{1-q}$, and the service time $\mathcal{X}$ follows an exponential distribution with an average of $\frac{1}{\mu_S}$.

Let T_S represent the processing latency of one key at Memcached servers, then T_S must be larger than the queueing time T_Q of the corresponding batch, but no larger than the completion time T_C of the batch, i.e.

$$T_Q < T_S \le T_C \tag{3}$$

Thus, we can estimate T_S by computing the queueing time and the completion time of the batch it belongs to.

The queueing time and the completion time of a batch can be obtained from a $GI/M/1$ model, which is transformed from the $GI^X/M/1$ model. This transformation has two steps.

- The inter-arrival gap of a batch follows the general distribution $\mathcal{T}_X(t)$ as above-mentioned.
- Suppose that a batch contains X keys indexed as $\{1, \cdots, X\}$ and the service time of the l_{th} key is $\mathcal{X}_l$. Hence, $\mathcal{X}_l$ follows the same exponential distribution of $\mathcal{X}$, and the batch size X follows a geometric distribution with a mean of $\frac{1}{1-q}$. The total service time of this batch $\mathcal{X}_X = \sum_{l=1}^{X} \mathcal{X}_l$ is a geometric sum of exponential stochastic variables. According to [32], $\mathcal{X}_X$ follows an exponential distribution with an average of $\frac{1}{(1-q)\mu_S}$. That is, the service time of a batch follows an exponential distribution with an average of $\frac{1}{(1-q)\mu_S}$.

Consequently, we can get CDF of the queueing time T_Q and the complete time T_C of a batch from this $GI/M/1$ queue, based on theorems in queuing theory [33],

$$T_Q(t) = 1 - \delta e^{-(1-\delta)(1-q)\mu_S t} \tag{4}$$

$$T_C(t) = 1 - e^{-(1-\delta)(1-q)\mu_S t} \tag{5}$$

where $\delta \in (0, 1)$ is the unique solution of

$$\delta = \mathcal{L}_{\mathcal{T}_X}\left((1-\delta)\mu_S\right) \tag{6}$$

and $\mathcal{L}_{\mathcal{T}_X}(s)$ denotes the Laplace transform of $\mathcal{T}_X(t)$. The value of δ is determined by the arrival pattern of keys and the server utilization. From (4) and (5), we can obtain the k^{th} quantile of T_Q and T_C, respectively,

$$(T_Q)_k = \max\left\{\frac{\ln\delta - \ln(1-k)}{(1-\delta)(1-q)\mu_S}, 0\right\} \tag{7}$$

$$(T_C)_k = \frac{-\ln(1-k)}{(1-\delta)(1-q)\mu_S} \tag{8}$$

Based on (3)(7)(8), we can get an estimation for the k^{th} quantile of T_S,

$$\max\left\{\frac{\ln\delta - \ln(1-k)}{(1-\delta)(1-q)\mu_S}, 0\right\} < (T_S)_k \le \frac{-\ln(1-k)}{(1-\delta)(1-q)\mu_S} \tag{9}$$

4.3.2 The maximum processing latency of N keys

According to our model (Section 3), there are a total of M Memcached servers. When an end-user request generates N keys, on average $p_j * N$ of them are hashed to Memcached server S_j $(j = 1, \cdots, M)$. Define

$$s_{i,j} \triangleq \begin{cases} s_i, & \text{if the } i_{th} \text{ request is distributed to server } S_i \\ 0, & \text{others} \end{cases}$$

Thus, $T_S(N) \triangleq \max\limits_{i=1,\cdots,N}\{s_i\} = \max\limits_{j=1,\cdots,M}\left\{\max\limits_{i=1,\cdots,N}\{s_{i,j}\}\right\}$. In this way, $T_S(N)$ can be considered as the maximum processing latency at the M Memcached servers.

Let T_{S_j} be the processing latency of one key at Memcached server S_j. Obviously, $s_{i,j}$ follows the same distribution as T_{S_j} as long as the i_{th} key is hashed to S_j. Thus, the CDF of $T_S(N)$ can be computed as follows.

$$\begin{aligned} T_S(N)(t) &= P\left\{\max_{j=1,\cdots,M}\left\{\max_{i=1,\cdots,N}\{s_{i,j}\}\right\} < t\right\} \\ &= \prod_{j=1}^{M}\prod_{i=1}^{N} P\{s_{i,j} < t\} \\ &= \prod_{j=1}^{M}[T_{S_j}(t)]^{P_j * N} \end{aligned} \tag{10}$$

where the second equal sign comes from the independence of $\{s_{i,j}\}$ for all $i = 1, \cdots, N$ and $j = 1, \cdots, M$, and the last equal sign comes from that there are on average $p_j * N$ keys hashed to the Memcached server S_j for each $j = 1, \cdots, M$. Define

$$T_S(1)(t) \triangleq \prod_{j=1}^{M}[T_{S_j}(t)]^{P_j} \tag{11}$$

$T_S(1)(t)$ can be treated as the CDF of a stochastic time $T_S(1)$, which is determined by T_{S_j} (the processing latency of one key at Memcached server S_j, $j = 1, \cdots, M$) and $\{p_j\}_{j=1}^M$ (the load distribution among Memcached servers). Combining (10) and (11), the CDF of $T_S(N)$ can be rewritten as

$$T_S(N)(t) = [T_S(1)(t)]^N$$

According to the approximation of the maximal statistics [34], the expectation of $T_S(N)$ can be approximated as the $\frac{N}{N+1}^{th}$ quantile of $T_S(1)$,

$$E[T_S(N)] = (T_S(1))_{\frac{N}{N+1}} \tag{12}$$

To obtain the $\frac{N}{N+1}^{th}$ quantile of $T_S(1)$, we get the following proposition after some algebraic derivations in Appendix 8.1.

Proposition 1. *Suppose that Memcached server S_1 has the heaviest load, i.e. p_1 is the maximum of $\{p_j\}_{j=1}^M$, then the k^{th} quantile of $T_S(1)$ is bounded by*

$$(T_{S_1})_{k^{\frac{1}{p_1}}} \leq (T_S(1))_k \leq (T_{S_1})_k \tag{13}$$

Proposition 1 indicates that latency for end-users is depended on the worst case among the Memcached servers, which is consistent with previous work [12, 13, 31]. With the bound of $(T_S(1))_k$ in (13) and the bound of $(T_S)_k$ in (9), the expectation of $T_S(N)$ in (12) is bounded by

$$\max\left\{\frac{\ln\delta + \ln(1-(\frac{N}{N+1})^{\frac{1}{p_1}})}{(1-\delta)(1-q)\mu_S}, 0\right\} \leq E[T_S(N)] \leq \frac{\ln(N+1)}{(1-\delta)(1-q)\mu_S} \tag{14}$$

4.4 Processing Latency at Database

In this part, we will estimate the processing latency of database, i.e. $T_D(N) = \max_{1\leq i\leq N}\{d_i\}$.

Let r denote the cache miss ratio and K be the number of missed keys out of total N keys. Then K follows a *multinomial distribution* with a mean of $N * r$. To estimate $T_D(N)$, we consider the following two cases.

1) $\boldsymbol{K = 0}$, *i.e. there are no missed keys.* This case has a probability of

$$P\{K = 0\} = (1-r)^N \tag{15}$$

And the expectation of $T_D(N)$ in this case is

$$E[T_D(N)|K = 0] = 0 \tag{16}$$

2) $\boldsymbol{K > 0}$*, i.e. there exist missed key.* This case has a probability of

$$P\{K > 0\} = 1 - (1 - r)^N \tag{17}$$

And the expectation of K in this case is

$$E[K|K > 0] = \frac{E[K]}{P\{K > 0\}} = \frac{N * r}{1 - (1 - r)^N} \tag{18}$$

Let T_D denote the processing latency of one missed key at database. According to our model (Section 3), we can obtain the CDF of T_D by solving an $M/M/1$ queue, where the service time follows an exponential distribution with an average of $\frac{1}{\mu_D}$, and the load is so light that the utilization $\rho \ll 1$,

$$T_D(t) = 1 - e^{-(1-\rho)\mu_D t} \approx 1 - e^{-\mu_D t} \tag{19}$$

For K missed keys, the processing latency at database d_i follows the same distribution of T_D. And for other keys, the processing latency at database d_i remains 0. Then the CDF of $T_D(N)$ can be computed as follows.

$$\begin{aligned} T_D(N)(t) &= P\left\{\max_{i=1,\cdots,N}\{d_i\} < t\right\} \\ &= \prod_{j=1}^{N} P\{d_i < t\} = [T_D(t)]^K \end{aligned} \tag{20}$$

According to the approximation of the maximal statistics [34], the expectation of $T_D(N)$ can be approximated by the $\frac{K}{K+1}^{th}$ quantile of T_D,

$$E[T_D(N)|K] \approx (T_D)_{\frac{K}{K+1}} = \frac{1}{\mu_D}\ln(K + 1) \tag{21}$$

where the last equal sign is because the CDF in (19). With (18) and (21), we can estimate the expectation of $T_D(N)$ in this case.

$$\begin{aligned} E[T_D(N)|K > 0] &\approx \tfrac{1}{\mu_D}\ln\left(E[K|K > 0] + 1\right) \\ &\approx \tfrac{1}{\mu_D}\ln\left(\tfrac{N*r}{1-(1-r)^N} + 1\right) \end{aligned} \tag{22}$$

Finally, combining the two cases, the expectation of $T_D(N)$ can be deduced as

$$\begin{aligned} E[T_D(N)] &= P\{K = 0\} * E[T_D(N)|K = 0] + \\ &\quad P\{K > 0\} * E[T_D(N)|K > 0] \\ &\approx \tfrac{1-(1-r)^N}{\mu_D} * \ln\left(\tfrac{N*r}{1-(1-r)^N} + 1\right) \end{aligned} \tag{23}$$

4.5 Latency Estimation

In summary, we obtain a latency estimation for the Memcached system.

Theorem 1. *The latency $T(N)$ of an end-user request that generates N Memcached keys is separately bounded by three parts,*

$$\max\{T_N(N), T_S(N), T_D(N)\} \le T(N) \le T_N(N) + T_S(N) + T_D(N)$$

1) The network latency $T_N(N)$ is almost constant;

2) The processing latency at Memcached servers $T_S(N)$ satisfies

$$\max\left\{\frac{\ln\delta + \frac{1}{p_1}\ln(N + 1)}{(1-\delta)(1-q)\mu_S}, 0\right\} \le E[T_S(N)] \le \frac{\ln(N + 1)}{(1-\delta)(1-q)\mu_S}$$

3) The processing latency at database $T_D(N)$ can be estimated by

$$E[T_D(N)] \approx \frac{1-(1-r)^N}{\mu_D} * \ln\left(\frac{N * r}{1-(1-r)^N} + 1\right)$$

Table 2: Factors that impact Latency in Memcached

Symbol	**Factor**
q	Concurrent probability of keys for each Memcached server.
μ_S	Average service rate at each Memcached server.
δ	Solution of (6). Determined by the arrival pattern of keys and the service rate at each Memcached server.
p_1	Largest load ratio among Memcached servers.
r	Cache miss ratio.
N	Number of keys generated from an end-user request

In Theorem 1, we prefer to estimate the latency for end-user requests in form of expected value rather than the $99.9th$ percentile value, because the expected value indicates the average level of the latency while the $99.9th$ percentile value only presents the bad case one end-user will experience with a tiny probability of 0.1%. In addition, the expected latency for an end-user request statistically equals to $\frac{N}{N+1}$ percentile of the latency for one Memcached key, which is one of the most concerned metric in recent performance evaluations [2, 31, 35]. As listed in Table 2, Theorem 1 reveals the main factors that impact latency in Memcached.

5 Validation and Quantitative Analysis

In this section, intensive experiments are conducted to validate our latency estimation in Theorem 1. Moreover, through these validations and numerical analysis, we obtain a quantitative understanding of how much improvement of latency performance can be achieved by optimizing each factor in Table 2.

5.1 Basic Validation

The first step is to show that Theorem 1 can exactly estimate the latency in Memcached under a realistic configuration. The testbed consists of two Memcached clients and four Memcached servers. Each client or server runs on a standalone physical machine with Intel® Core™ i5-5200U CPU and 8GB memory. All machines connect with each other through a ToR switch and all links are 1Gbps. A total of 512 connections are created between the clients and servers. We execute the multi-thread Memcached test tool of *mutilate* [36], which has been well-tested [16, 35], and generate workload according to Section 5 of [3], which provides a statistical model based on the real Facebook trace. In detail, the arriving keys for each Memcached server have a concurrent probability $q = 0.1$, and the interarrival time between adjacent keys follows a Generalized Pareto distribution

$$\mathcal{T}_X(t) = 1 - \left(1 + \frac{\xi\lambda t}{1-\xi}\right)^{-1/\xi} \tag{24}$$

where the average arrival rate $\lambda = 62.5Kps$ and the burst degree $\xi = 0.15$. In our deployment, the average service rate at each Memcached server μ_S and the network latency n_i are measured to be $80Kps$ and $50\mu s$, respectively. We configure that each end-user request generates $N = 150$ Memcached keys, the load distribution among all Memcached servers is balanced, the cache miss ratio $r = 0.01$, and the average service time at database $\mu_D^{-1} = 1ms$.

The experiment lasts for 10 min and generates almost 10^6 end-user requests in total. We measure the average end-user request latency $T(N)$, as well as three partial latencies, i.e. the network latency $T_N(N)$, the processing latency at Memcached servers $T_S(N)$, and the processing latency at database $T_D(N)$. As shown in Table 3, the experiment results are matched with the latency estimation in Theorem 1. Moreover, we record T_S, the processing latency for a single key at Memcached servers. As drawn in *Fig.*4, the k^{th} quantile of T_S is tightly bounded by (9). That is, in the case of Facebook workload, our estimation about the Memcached latency is correct.

Table 3: Validating Facebook Workload

Latency	Theorem 1	Experiment	Confidence Interval
$T_N(N)$	$20\mu s$	$20\mu s$	$[18.12\mu s, 21.68\mu s]$
$T_S(N)$	$351\mu s \sim 366\mu s$	$368\mu s$	$[362\mu s, 373\mu s]$
$T_D(N)$	$836\mu s$	$867\mu s$	$[855\mu s, 879\mu s]$
$T(N)$	$836\mu s \sim 1222\mu s$	$1144\mu s$	$[1128\mu s, 1160\mu s]$

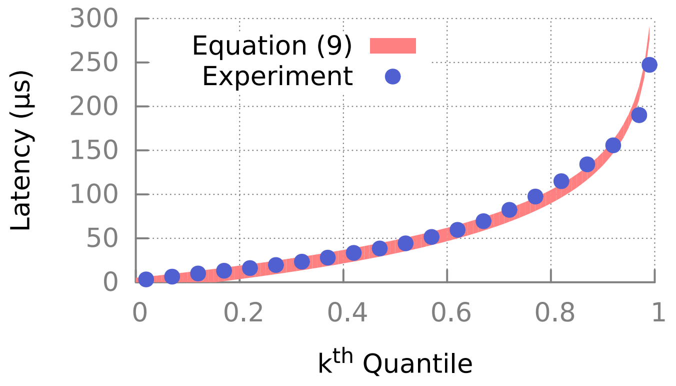


Figure 4: The k^{th} quantile of processing latency for one key at Memcached servers.

5.2 Impacts of Factors

Subsequently, we will show that Theorem 1 accurately estimates the latency in Memcached under all actually depolyable configurations. Configurable factors include *the workload pattern*, *the load distribution among Memcached servers*, *the number of keys generated from an end-user request*, and *the cache miss ratio*. At the same time, the improvement of latency performance achieved by optimizing each factor is quantitated. When discussing one factor, others maintain the original values as in Section 5.1.

5.2.1 Workload Pattern

The workload pattern mainly impacts the processing latency $T_S(N)$ at Memcached servers. And there are three specific factors in the workload pattern, i.e. *the concurrent probability of keys*, *the burst degree of keys* and *the average arrival rate of keys*.

i) q**: The concurrent probability of keys.** We repeat the experiment in Section 5.1 with the concurrent probability of keys (q) varying from 0 to 0.5 and record the average value of $T_S(N)$. As illustrated in *Fig.*5, the experiment results of $E[T_S(N)]$ is very close to that computed with Theorem 1.

Obviously, the more concurrently keys arrive, the larger processing latency will be. In our model, concurrent keys are treated as batches. The average batch size $\frac{1}{1-q}$ indicates how many keys concurrently arrive at one Memcached server in the average sense. According to the estimation of $T_S(N)$ in (14), $E[T_S(N)]$ grows linearly with the increase of $\frac{1}{1-q}$, i.e.

$$E[T_S(N)] = \Theta\left(\frac{1}{1-q}\right)$$

ii) ξ**: The burst degree of keys**. In the Generalized Pareto distribution, a larger ξ indicates more burst arrivals and $\xi = 0$ means that the arrival pattern is Poisson. We choose different ξ from 0 to 0.6 and repeat the experiment in Section 5.1. As drawn in *Fig.*6, the experiment result of $E[T_S(N)]$ is exactly matched with that of Theorem 1.

The result shows that the more burst the keys, the higher processing latency end-users will experience. According to the estimation of $T_S(N)$ in (14), the quantitative relationship between $T_S(N)$ and ξ is implicit in the value of δ. And we will provide an in-depth discussion about δ later.

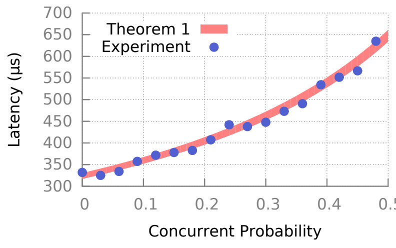


Figure 5: Evolution of $E[T_S(N)]$ when q varies from 0 to 0.5.

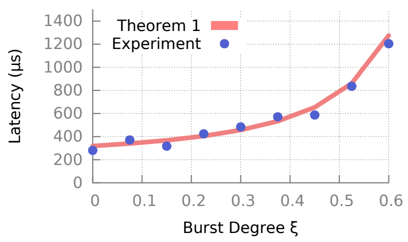


Figure 6: Evolution of $E[T_S(N)]$ when ξ varies from 0 to 0.6.

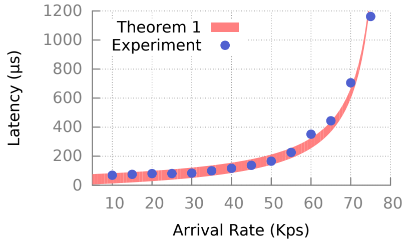


Figure 7: Evolution of $E[T_S(N)]$ when λ varies from $10Kps$ to $75Kps$.

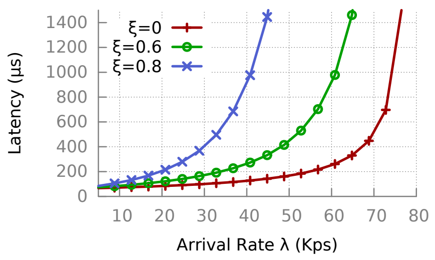


Figure 8: Evolution of $E[T_S(N)]$ when $\xi = 0, 0.6, 0.8$, λ varies from $10Kps$ to $75Kps$ and $\mu_S = 80Kps$.

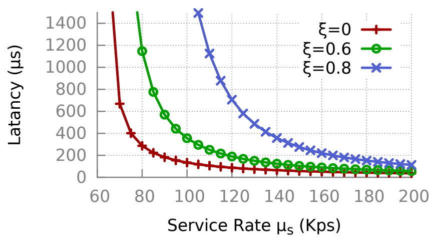


Figure 9: Evolution of $E[T_S(N)]$ when $\xi = 0, 0.6, 0.8$, $\lambda = 62.5Kps$ and μ_S varies from $65Kps$ to $200Kps$.

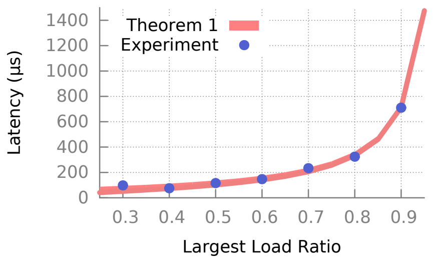


Figure 10: Evolution of $E[T_S(N)]$ when p_1 varies from 0.1 tp 0.9, $\Lambda = 80Kps$ and $\mu_S = 80Kps$.

iii) **λ: The average arrival rate of keys.** With the arrival rate λ varying from $10Kps$ to $75Kps$, the experiment in Section 5.1 is repeated and the evolution of $E[T_S(N)]$ is recorded. As depicted in *Fig*.7, the experiment result is quite close to that of Theorem 1.

From *Fig*.7, we can find $E[T_S(N)]$ increases gently when λ is less than $50Kps$, but sharply as λ grows more than $60Kps$. Therefore, there exists a cliff point of latency when λ is about $60Kps$. At this cliff point, the corresponding Memcached server utilization $\rho_S \triangleq \frac{\lambda}{\mu_S}$ is about 75%.

Similar to ξ, the quantitative relationship between $T_S(N)$ and λ is implicit in the value of δ in (14). To achieve an in-depth understanding of the cliff point, we conduct a numerical analysis on the special parameter δ.

Discussion: According to (6), the value of δ is determined by not only ξ and λ, but also the average service rate at Memcached servers μ_S. With the variation of ξ, λ and μ_S, $E[T_S(N)]$ evolutes regularly according to Theorem 1.

Fig.8 shows the evaluations of $E[T_S(N)]$ in Theorem 1, when ξ respectively takes the value of 0, 0.6, 0.8, λ varies from $10Kps$ to $75Kps$, and $\mu_S = 80Kps$. When key arrivals become more burst, $E[T_S(N)]$ reaches the cliff point at a lower λ. When $\xi = 0$, $E[T_S(N)]$ reaches the cliff point when λ is more than $65Kps$ and ρ_S is as high as 80%. However, when $\xi = 0.6$ and 0.8, $E[T_S(N)]$ gets to the cliff point when λ are just $45Kps$ and $30Kps$, whose corresponding ρ_S are only 55% and 40%, respectively.

Fig.9 illustrates the evaluations of $E[T_S(N)]$ in Theorem 1, when $\xi = 0, 0.6, 0.8$, $\lambda = 62.5Kps$ and μ_S varies from $65Kps$ to $200Kps$. There also exists a cliff point of $E[T_S(N)]$ when μ_S grows to a specific value. And the more burst keys lead to a higher μ_S when $E[T_S(N)]$ reaches the cliff point. When $\xi = 0$, $E[T_S(N)]$ reduce sharply when μ_S varies from $65Kps$ to $80Kps$, but much slower and gently when μ_S increases more than $90Kps$. Thus, $E[T_S(N)]$ reaches the cliff point when μ_S is about $85Kps$ and $\rho_S = 80\%$. Moreover, when $\xi = 0.6$ and 0.8, the cliff point of $E[T_S(N)]$ is delayed when μ_S are as high as $110Kps$ and $160Kps$, whose corresponding ρ_S are only 55% and 40%, respectively.

It is noteworthy that as long as the burst degree ξ remains unchanged, the server utilization ρ_S at the cliff point will also keep constant, no matter what value λ takes. Based on (6), we have the following proposition after some algebraic derivations in Appendix 8.2,

Proposition 2. *The processing latency at Memcached servers reaches a cliff point when the Memcached server*

Table 4: Upper Bound for Server Utilization

ξ	$\rho_S(\xi)$	ξ	$\rho_S(\xi)$	ξ	$\rho_S(\xi)$	ξ	$\rho_S(\xi)$
0.00	77%	0.25	73%	0.50	65%	0.75	45%
0.05	76%	0.30	72%	0.55	62%	0.80	39%
0.10	76%	0.35	71%	0.60	59%	0.85	31%
0.15	75%	0.40	69%	0.65	55%	0.90	21%
0.20	74%	0.45	67%	0.70	50%	0.95	9%

utilization gets to a specific value. This value is only determined by the burst degree of keys.

This finding provides an upper bound for server utilization when configuring a Memcached system. As listed in Table 4, we estimate some specific server utilization $\rho_S(\xi)$ at the cliff point of latency under different burst degree ξ. To ensure low latency, we recommend that the server utilization ρ_s should not extend $\rho_S(\xi)$ if the key arrivals have a burst degree ξ.

5.2.2 The load distribution among Memcached servers

The load distribution among Memcached servers mainly impacts $T_S(N)$, the processing latency at Memcached servers. We generate a key stream at the rate of $\Lambda = 80Kbps$ and distribute these keys to each Memcached server. In the meanwhile, we vary the largest load ratio among Memcached servers p_1 from 0.3 to 0.9. And we set the burst degree $\xi = 0.15$ and the average service rate at each Memcached server $\mu_S = 80Kps$. That is, the heaviest Memcached server has to hold on a load of $p_1 * \Lambda$ and the corresponding server utilization $\rho_S = \frac{p_1 * \Lambda}{\mu_S}$. As illustrated in *Fig.*10, the average value of T_S is tightly bounded by the estimation of $T_S(N)$ in (14).

Apparently, $T_S(N)$ reaches a cliff point when $p_1 = 0.75$, i.e. the largest load and the server utilization among Memcached servers are respectively $60Kps$ and 75%. This result is consistent with Table 4. Accurately, Proposition 2 is also tenable when the load distribution among Memcached servers is unbalanced.

i) If the largest utilization among all the Memcached servers is lower than $\rho_S(\xi)$, the processing latencies at different servers are almost the same. In this case, the load-balancing mechanisms are unnecessary.

ii) On the opposite, once the heaviest load among all the Memcached servers is larger than $\rho_S(\xi)$, the worst latency of all servers will become severe. In this case, the load-balancing mechanisms should take effect.

That is, the specific server utilization $\rho_S(\xi)$ provides a novel guideline for load-balancing mechanisms.

5.2.3 The cache miss ratio

The tiny cache miss ratio r mainly impacts the processing latency at database $T_D(N)$. To evaluate such impact, we vary the cache miss ratio r from 10^{-4} to 10^{-1}, and record $E[T_D(N)]$ under each r value. *Fig.*11 depict the experiment results as well as the theoretical value calculated according to Theorem1 . Although there is slight difference between Theorem1 and experiment results, they present the same trend: i) If an end-user request generates a small number of Memcached keys, $E[T_D(N)]$ grows linearly with the increase of r. ii) If an end-user request generates abundant Memcached keys, $E[T_D(N)]$ grows logarithmically with the increase of r.

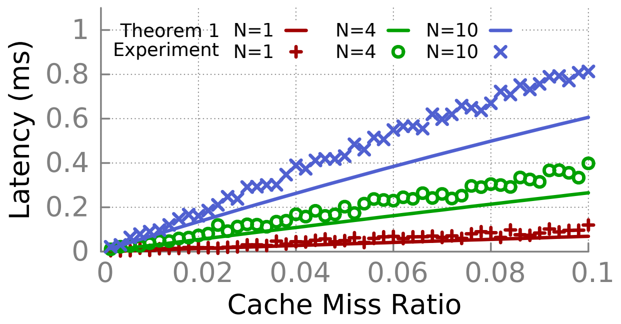
Based on the estimation of $T_D(N)$ in (23), this complex relationship between $T_D(N)$ and r is proved on Appendix 8.3 and briefly expressed as follows,

$$E[T_D(N)] = \begin{cases} \Theta(r), & \text{if } N \text{ is small} \\ \Theta(\log r), & \text{if } N \text{ is large} \end{cases} \tag{25}$$

This finding can be explained as follows.

i) If there are only a few keys generated from an end-user request, there exists a large probability that there are no missed keys. In this case, the processing latency at database depends on whether there exist missed keys to a large extent. Therefore, reducing the cache miss ratio can avoid the large cost of missing and significantly improve latency.

ii) If there are numerous keys generated from an end-user request, it will be inevitable that there are missed keys. Thus, the processing latency of database is mainly dependent on how many keys are missed, in which case reducing the cache miss ratio just avoids missing more keys but can not avoiding the huge cost of missing.



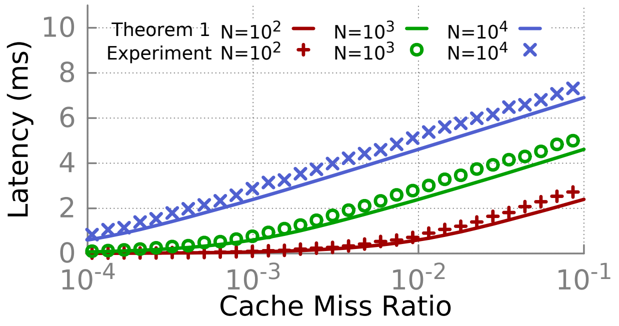


Figure 11: Evolution of $E[T_D(N)]$ when the cache miss ratio r varies from 10^{-4} to 10^{-1}.

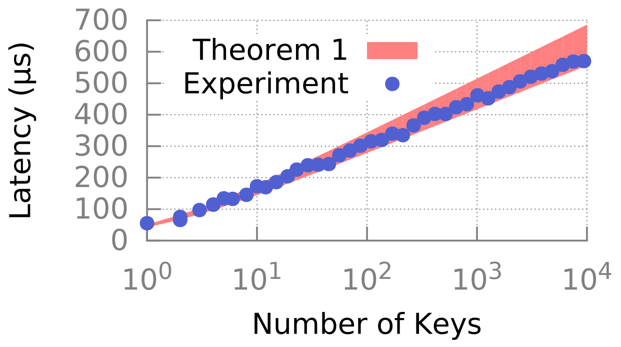


Figure 12: Evolution of $E[T_S(N)]$ when an end-user request generates $1 \sim 10^4$ keys.

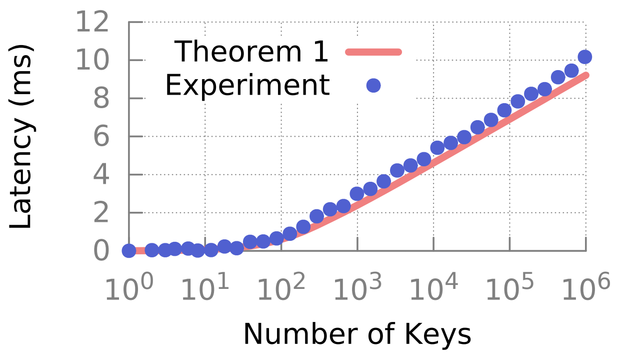


Figure 13: Evolution of $E[T_D(N)]$ when an end-user request generates $1 \sim 10^6$ keys.

Generally, the number of keys generated from an end-user request is always huge, that is, the latter case is more common. Therefore, the improvement of latency is only logarithmic with the decrease of cache miss ratio.

5.2.4 The number of keys generated from an end-user request

The number of keys N impacts both the processing latency at Memcached server $T_S(N)$ and the processing latency at database $T_D(N)$. As drawn in *Fig.*12 and *Fig.*13, the experiment results are exactly matched with Theorem 1 when an end-user request generates $1 \sim 10^4$ keys. Both $E[T_S(N)]$ and $E[T_D(N)]$ seem to grow logarithmically when an end-user request generates more keys.

The logarithmic relationship between $T_S(N)$ and N is obvious from the estimation of $T_S(N)$ in (14),

$$E[T_S(N)] = \Theta(\log N)$$

And the logarithmic relationship between $T_S(N)$ and N can be obtained after some derivations on the estimation of $T_S(N)$ in (23).

$$\begin{aligned} & \lim_{N\to\infty} E[T_D(N)] \\ \approx & \lim_{N\to\infty} \frac{1-(1-r)^N}{\mu_D} * \ln\left(\frac{N*r}{1-(1-r)^N}+1\right) \\ = & \lim_{N\to\infty} \frac{1}{\mu_D} * \ln\left(N*r+1\right) \end{aligned}$$

That is,

$$E[T_D(N)] = \Theta(\log N)$$

Consequently, we demonstrate that the end-user latency $T(N)$ also grows logarithmically with the increase of N. Similar results have been proved in previous work [14] for MapReduce.

5.3 Insights and Recommendations

This section not only validates that Theorem 1 exactly estimates the latency in Memcached with a large range of configurations, but also quantitates the impact of each factor on latency in Memcached. The quantitative understanding further provides useful recommendations for optimizing latency in Memcached.

1) **Workload and service rate at Memcached servers.** We find that the processing latency in Memcached servers reaches a cliff point when the Memcached server utilization gets to a specific value. This value is only determined by the burst degree of keys and typically equals 75% under the Facebook workload. Different from previous work that improves latency performance by reducing the load [2] and increasing the service rate of Memcached servers as much as possible [4, 5, 8–11, 17, 18], we recommend the server utilization should not exceed a specific value to ensure low latency.

2) **Load distribution among Memcached servers.** We find that the processing latency of Memcached servers becomes severe when the largest utilization exceeds the specific utilization at the latency cliff point. Consequently, we recommend the load-balancing mechanisms just to take effect before the largest utilization extends the specific value.

3) **Cache miss ratio and the number of keys per request.** We find that only when one end-user request generates a small number of Memcached keys, the impact of the cache miss ratio is significant. But in the common case that an end-user request generates numerous Memcached keys, the latency is mainly based on the number of Memcached keys and the improvement of latency performance is only logarithmic with the decreasing of the cache miss ratio. At the same time, we also find that the end-user latency grows logarithmically with the increase of the number of keys per request. Since an end-user request always generates numerous Memcached keys and the cache miss ratio is always quite tiny, we prefer to recommend drastically minimizing the number of Memcached keys generated from each end-user request, instead of reducing the tiny cache miss ratio.

6 Conclusions

This paper aims to quantitate the impact of different factors on latency in the Memcached system. We establish a specific model for Memcached and derive a estimation about the latency. The latency estimation is validated by sufficient experiments. We find that the main factors that impact latency in Memcached include the workload pattern and the service rate at Memcached servers, the load distribution among Memcached servers, the cache miss ratio and the number of Memcached keys. The key insights and recommendations in this paper are twofold:

1) The processing latency at Memcached Servers reaches a cliff point when the server utilization gets to a specific value, which is negatively related with the burst degree of keys. Under the Facebook workload, the specific utilization is about 75%. To ensure low latency, we recommend that the Memcached server utilization should keep below the specific utilization, and load-balancing mechanisms should take effect before and Memcached server extends the specific utilization.

2) The latency grows logarithmically, when an end-user request generates more Memcached keys or the cache missing ratio increases. Since an end-user request always generates numerous Memcached keys and the cache miss ratio is always quite tiny, we prefer to recommend minimizing the number of Memcached keys generated from an end-user request by a drastic degree to improve latency, rather than reducing the tiny cache miss ratio.

7 Acknowledgments

The authors gratefully acknowledge our shepherd Mr. Anshul Gandhi and the anonymous reviewers for their constructive comments. This work is supported in part by National High-Tech Research and Development Plan of China (863 Plan) under Grant No.2015AA020101, National Natural Science Foundation of China (NSFC) under Grant No.61225011 and No.61502539, Suzhou-Tsinghua Special Project for Leading Innovation, and China Postdoctoral Science Foundation under Grant No.2015M582344 and No.2016T90761.

8 Appendix

8.1 Proof of Proposition 1

Proof. i) Set $t_1 = (T_{S_1})_{k^{\frac{1}{p_1}}}$, then

$$\begin{cases} T_{S_1}(t_1) = k^{\frac{1}{p_1}}, \\ T_{S_j}(t_1) \le 1, \qquad j = 2, \cdots, M \end{cases} \tag{26}$$

Combing (11) and (26), we have

$$T_S(1)(t_1) = \prod_{j=1}^{M} [T_{S_j}(t_1)]^{P_j} \le [T_{S_1}(t_1)]^{P_1} = k$$

Therefore, $(T_S(1))_k \ge (T_{S_1})_{k^{\frac{1}{p_1}}}$.

ii) Set $t_2 = (T_{S_1})_k$, then

$$\begin{cases} T_{S_1}(t_2) = k, \\ T_{S_j}(t_2) = \frac{N}{N+1}, \quad j = 2, \cdots, M \end{cases} \tag{27}$$

Combing (11) and (27), we have

$$T_S(1)(t_2) = \prod_{j=1}^{M} [T_{S_j}(t_2)]^{P_j} \ge \prod_{j=1}^{M} [k]^{p_j} = k$$

Therefore, $(T_S(1))_{\frac{N}{N+1}} \le (T_{S_1})_k$. □

8.2 Proof of Proposition 2

Proof. Supposing that there are two Memcached servers S_1 and S_2, the request arrival rates on the two servers are λ_1 and λ_2 satisfying

$$\lambda_1 = c\lambda_2$$

where c is a positive constant. And the inter-arrival gaps $\mathcal{T}_{X_1}$ and $\mathcal{T}_{X_2}$ follow similar distributions, i.e.

$$\mathcal{T}_{X_1}(t) = \mathcal{T}_{X_2}(ct)$$

If they have the same utilization, i.e. $\frac{\lambda_1}{\mu_{S_1}} = \frac{\lambda_2}{\mu_{S_2}}$, we have

$$\mu_{S_1} = c\mu_{S_2}$$

Consequently, we can calculate δ_1 and δ_2 according to (6),

$$\begin{aligned} \delta_1 &= \mathcal{L}_{\mathcal{T}_{X_1}}((1-\delta_1)(1-q)\mu_{S_1}) \\ &= \int_0^\infty e^{-(1-\delta_1)(1-q)\mu_{S_1} t} d\mathcal{T}_{X_1}(t) \\ &= \int_0^\infty e^{-(1-\delta_1)(1-q)c\mu_{S_2} t} d\mathcal{T}_{X_2}(ct) \\ &= \int_0^\infty e^{-(1-\delta_1)(1-q)\mu_{S_2} t} d\mathcal{T}_{X_2}(t) \\ &= \mathcal{L}_{\mathcal{T}_{X_2}}((1-\delta_1)(1-q)\mu_{S_2}) \end{aligned}$$

Note that δ_2 is the unique solution of

$$\delta = \mathcal{L}_{\mathcal{T}_{X_2}}((1-\delta)(1-q)\mu_{S_2})$$

we have

$$\delta_1 = \delta_2$$

Consquently, according to (14), we have

$$E[T_{S_1}(N)] = \frac{1}{c} E[T_{S_2}(N)]$$

Thus, the two servers have similar latency evolutions while the utilization changes and have a same cliff utilization. □

8.3 Proof of Equation (25)

Proof. 1) If N is small,

$$\begin{aligned} & E[T_D(N)] \\ \approx\ & \frac{1-(1-r)^N}{\mu_D} * \ln\left(\frac{N*r}{1-(1-r)^N}+1\right) \\ =\ & \frac{N*r+o(r^2)}{\mu_D} * \ln\left(\frac{1}{1+o(r)}+1\right) \\ =\ & \Theta(r) \end{aligned}$$

2) If N is large,

$$\begin{aligned} & E[T_D(N)] \\ \approx\ & \lim_{N\to\infty} \frac{1-(1-r)^N}{\mu_D} * \ln\left(\frac{N*r}{1-(1-r)^N}+1\right) \\ =\ & \lim_{N\to\infty} \frac{1}{\mu_D} * \ln\left(N*r+1\right) \\ =\ & \Theta(log(N)) \end{aligned}$$

Therefore,

$$E[T_D(N)] = \begin{cases} \Theta(r), & \text{if } N \text{ is small} \\ \Theta(\log r), & \text{if } N \text{ is large} \end{cases}$$

□

References

[1] "Memcached - a distributed memory object caching system." http://memcached.org/.

[2] 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 *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

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

[4] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bár, and Z. István, "Achieving 10gbps line-rate key-value stores with fpgas," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[5] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing soc accelerators for memcached," 2013.

[6] W. Zhang, J. Hwang, T. Wood, K. Ramakrishnan, and H. Huang, "Load balancing of heterogeneous workloads in memcached clusters," in *9th International Workshop on Feedback Computing (Feedback Computing 14)*, 2014.

[7] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: scaling performance cliffs in web memory caches," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[8] N. S. Islam, M. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance rdma-based design of hdfs over infiniband," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[9] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda, "Scalable memcached design for infiniband clusters using hybrid transports," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, 2012.

[10] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur *et al.*, "Memcached design on high performance rdma capable interconnects," in *2011 International Conference on Parallel Processing*, 2011.

[11] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy nodes with 10gbe: leveraging one-sided operations in soft-rdma to boost memcached," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[12] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low latency via redundancy," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, 2013.

[13] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[14] A. Rizk, F. Poloczek, and F. Ciucu, "Computable bounds in fork-join queueing systems," in *ACM SIGMETRICS Performance Evaluation Review*, 2015.

[15] D. Gamarnik, J. N. Tsitsiklis, and M. Zubeldia, "Delay, memory, and messaging tradeoffs in distributed service systems," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, 2016.

[16] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[17] A. Wiggins and J. Langston, "Enhancing the scalability of memcached." http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached, 2012.

[18] "Configuration and deployment guide for memcached on intel architecture." https://software.intel.com/sites/default/files/ article/402153/configuration-and-deployment-guide-for-memcached-on-intel.pdf.

[19] C. Li and A. L. Cox, "Gd-wheel: a cost-aware replacement policy for key-value stores," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[20] L. Cherkasova, *Improving WWW proxies performance with greedy-dual-size-frequency caching policy.* Hewlett-Packard Laboratories, 1998.

[21] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: Dynamic cloud caching," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[22] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[23] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[24] H. Bjornsson, G. Chockler, T. Saemundsson, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013.

[25] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Communications of the ACM*, 2008.

[26] "Amazon elastic compute cloud ec2." http://aws.amazon.com/ec2.

[27] S. Varma and A. M. Makowski, "Interpolation approximations for symmetric fork-join queues," *Performance Evaluation*, 1994.

[28] R. Nelson and A. N. Tantawi, "Approximate analysis of fork/join synchronization in parallel queues," *IEEE transactions on computers*, 1988.

[29] A. S. Lebrecht and W. J. Knottenbelt, "Response time approximations in fork-join queues," in *23rd UK Performance Engineering Workshop (UKPEW)*, 2007.

[30] S.-S. Ko and R. F. Serfozo, "Sojourn times in g/m/1 fork-join networks," *Naval Research Logistics (NRL)*, 2008.

[31] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[32] R. Nelson, "Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling," 2013.

[33] J. Medhi, *Stochastic models in queueing theory.* Academic Press, 2002.

[34] G. Casella and R. L. Berger, *Statistical inference*, 2002.

[35] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughput and low latency," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[36] "Mutilate: high-performance memcached load generator." https://github.com/leverich/mutilate.