Chisel

Reshaping Queries to Trim Latency in Key-Value Stores

Birke, Robert; Perez, Juan F.; Mokhtar, Sonia Ben; Rameshan, Navaneeth; Chen, Lydia Y.

DOI
10.1109/ICAC.2019.00016

Publication date
2019

Document Version
Final published version

Published in
Proceedings - 2019 IEEE International Conference on Autonomic Computing, ICAC 2019

2019 IEEE International Conference on Autonomic Computing (ICAC)

Chisel: Reshaping Queries to Trim Latency in Key-Value Stores

Robert Birke*, Juan F. Pérez†, Sonia Ben Mokhtar‡, Navaneeth Rameshan§, Lydia Y. Chen¶
*ABB Corporate Research, Baden-Dättwil, Switzerland. Email: robert.birke@ch.abb.com
†Universidad del Rosario, Bogotá, Colombia. Email: juanferna.perez@urosario.edu.co
‡INSA Lyon, Lyon, France. Email: sonia.benmokhtar@insa-lyon.fr
§IBM Research Zurich, Rúschlikon, Switzerland. Email: nav@zurich.ibm.com
¶TU Delft, Delft, Netherlands. Email:y.chen-10@tudelft.nl

***Abstract*—It is challenging for key-value data stores to trim user (tail) latency of requests as the workloads are observed to have skewed number of key-value pairs and commonly retrieved via multiget operation, i.e., all keys at the same time. In this paper we present Chisel, a novel client side solution to efficiently reshape the query size at the data store by adaptively splitting big requests into chunks to reap the benefits of parallelism and merge small requests into a single query to amortize latency overheads per request. We derive a novel layered queueing model that can quickly and approximately steer the decisions of Chisel. We extensively evaluate Chisel on memcached clusters hosted on a testbed, across a large number of scenarios with different workloads and system configurations. Our evaluation results show that Chisel can overturn the inherent high variability of requests into a judicious operational region, showcasing significant gains for the mean and 95^{th} percentile of user perceived latency, compared to the state-of-art query processing policy.**

I. Introduction

Key-value data stores, such as memcached, are widely deployed to scale up the performance of distributed services in production systems, e.g., Facebook [1] and Twitter [2]. Their popularity is grounded on their speed to serve user requests few order of magnitude faster than querying back-end databases or accessing file systems.

Users' requests for interactive web services display a highly varying degree of fanout [1, 3], intensifying many times the challenge of delivering consistent latency. For example, although the average number of keys in a single request is roughly 24 at Facebook [1], around 10% of requests ask for more than 100 keys (elephant request), showing a skewed number of key-value pairs per user request. To cater to users' requests asking for several data elements, multiget APIs are offered to batch multiple read operations, further propagating the skewed request sizes in key-value stores.

It is known that latency, particularly its tail, can drastically degrade due to highly varying sizes as elephant requests/queries require long processing times and cause long waiting times for any *mice* requests behind them. While a significant number of prior studies [4, 5, 6] try to minimize the key-value retrieval time for single key-value pairs via well-engineered implementations and intelligent load balancing [7, 5], little is known on managing latency for workloads with a skewed number of key-value queries per request. Carefully scheduling multiget requests [3] has been shown effective in minimizing the tail latency, reducing the difference among operations' finishing times.

Another dimension of multiget requests is that resource bottlenecks can switch depending on the mix of request sizes, how queries are processed at the datastores, and the servers' load. On the one hand, processing big requests as small parallel queries can reduce the processing time but the overhead of excessive parallelization increases the risk of server overloads and stragglers [8, 9, 10][1]. In fact, determining optimal parallelism levels is a long standing challenge [11]. On the other hand, merging small requests can reduce the overhead, increasing the throughput in retrieved key-value pairs at the expense of higher latency, particularly important during high load.

In this paper, we propose a novel client-side solution, Chisel, which adaptively and proactively splits and merges user requests into queries to retrieve key-value pairs. Particularly, Chisel splits big elephant requests into smaller parallel queries and merges small mice requests by piggybacking them during the connection setup time. Chisel is a drop-in solution that can be deployed on multiple clients. The key components of Chisel are an analyzer thread, a dispatcher thread, and a pool of query threads that enable non-blocking and parallel processing of requests and queries. The analyzer thread determines the optimal split and merge levels based on a set of novel layered queueing models that capture the complex interplay among request arrivals, the loads on clients and memcached servers, and most importantly the interaction among split and merge operations.

[1]In the following we refer to request as the original demand to the datastore client and to query as the effective command sent to the datastore server.

2474-0756/19/$31.00 ©2019 IEEE
DOI 10.1109/ICAC.2019.00016

We particularly focus on memcached to demonstrate the effectiveness of Chisel and two prototype implementations, i.e., using the go programming language and a more efficient one using libevent. Our extensive evaluations show that Chisel can achieve significant latency reductions for both the mean and 95th percentile latency over hundreds of scenarios, respectively, compared to the standard practice of processing all keys of a request as a single memcached query.

The main contributions of Chisel are:,

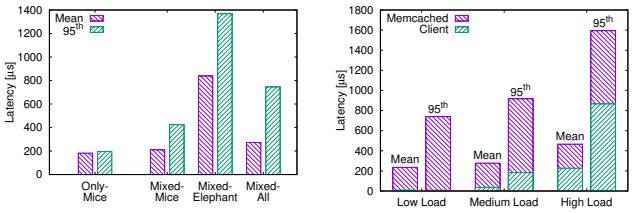
- Chisel can reshape the query sizes by intelligently combining split and merge operations. It is particularly effective to improve the (tail) latency by reducing the request skew.
- Chisel takes a holistic approach that accounts for latency at both the memcached client and server, and is able to reduce the overall latency by trading-off the benefits of parallelism with its additional overhead;
- Chisel is able to adapt to the varying workload and system conditions, appropriately adjusting the split and merge levels without intrusive modifications to the memcached servers.

II. Background and Motivation

User requests in web-serving architectures are typically served by a client that acts as an interface to a back-end data store. User-perceived latency is thus composed of the time spent at the client and the remote time to retrieve the objects from the data store. The time spent within the client encompasses library overhead, waiting time and processing time if any. Processing time depends on the role of the client. If the client simply acts as a proxy for forwarding requests, the processing time is almost non-existent. If the client is not just a proxy but provides some management actions or performs some pre-processing, then the client time includes a non-negligible processing time. The remote time includes the time spent at the data store and the transfer times at the network time. Both these times grow linearly with the number of objects requested.

Request Skew. The size of users' requests in terms of number of objects (keys in a key-value store) has been shown to be highly skewed in production systems. For example, the average number of keys per user request at Facebook [1] is 24 but the 95th percentile is more than 95 keys. Another study on SoundCloud [3] identifies that 40% of the requests have only one key while the 99th percentile is around 100 keys. This is essentially the scenario of elephant and mice requests: a large number of requests asks for few keys, while a small percentage of requests asks for a large number of keys.

Batching. The state of practice is to retrieve all the objects in one query via multiget API, also known as



(a) Homogeneous v.s. skewed (b) Time breakdown

Fig. 1: The performance baseline: skewed requests degrade performance and the bottlenecks alter.

batching, and is widely used in today's data stores, e.g., Cassandra, MongoDB, Redis, and Memcached [2]. *Batching* essentially reduces the overall latency by avoiding the network RTT associated with querying each object, as requests for multiple objects are aggregated within a single query. For in-memory data stores such as memcached, networking is often the most expensive part of a memcached request (Quora reports network time accounts for over 80% of the total time processing Memcached requests [12]) and batching is employed widely to reduce network latency.

A. Impact of Skewed Workloads

We now demonstrate two key performance aspects using memcached as an example: the impact of skewed workloads and the performance bottleneck shift between client and server depending on the offered load. To this end we perform experiments, using batching, on a memcached replica set of 4 servers hosted on our testbed, detailed in Section V-A. To capture the impact of skewed workloads we setup the following scenarios: (i) homogeneous *mice* requests asking for 10 keys, and (ii) skewed requests where mice (90%) ask for 10 keys and elephants (10%) ask for 100 keys. To make a fair comparison, we set the mean inter-arrival time to 100 and 190 μs for homogeneous and skewed requests, respectively, keeping the same mean number of keys requested per second in both scenarios.

Skewed workloads degrade performance. Fig. 1(a) summarizes the mean and 95th percentile latency under skewed and homogeneous workloads. In the skewed request scenario, we show the overall latency and further decompose it into the latency experienced by mice and elephant requests individually. We can clearly see that the 95th percentile latency of mice degrades by a factor of 2.1X in the skewed scenario, compared to the homogeneous case. This is because mice requests experience longer queueing times, waiting behind elephant requests. Further, the level of degradation increases with the load, as the probability of waiting behind elephant requests

[2]Throughout this paper, we refer to requests as the workload unit from the user to the client and query as the workload unit from the client to the data store.

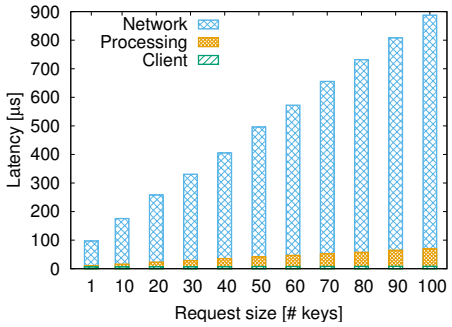


Fig. 2: Batching multiple keys into one query: breakdown of user perceived latency into client and memcached that includes processing and network time.

is higher. Moreover, because of elephant requests, the *overall* mean and 95^{th} percentile latency increases by a factor of 1.5X and 3.8X, respectively. In a nutshell, the skewed workload worsens the performance with respect to the homogenous workload, especially hurting the mice requests and the tail latency.

Resource bottlenecks shift. We illustrate how the performance bottleneck shifts between clients and servers at different loads, when the client acts as more than just a proxy. We use a skewed workload as above (90% mice and 10% elephant) at mean inter-arrival times of 400, 200 and 100 μs, resulting into low, medium and high loads at the client. We compute the latency breakdown between the time spent locally in the memcached client library (termed client time hereon) and remotely at the memcached server (termed memcached time hereon), which includes the network transfer time. Fig. 1(b) summarizes the mean and 95^{th} percentile under the three different loads. For low and medium loads, the *memcached time* accounts for the majority of the mean and tail latency, representing the bottleneck; whereas for high load the *client time* grows and becomes the bottleneck. This shift in the bottleneck resource points to different opportunities for latency improvement. For example, one shall try to reduce memcached time during low loads, for instance by exploiting parallel processing of the keys within a request, whereas one shall try to minimize the overhead at the client during high loads, for instance by merging keys from multiple requests.

B. The Limits of Batching

As we have seen in the previous section *batching* suffers a disadvantage in processing skewed requests. Using an example, we demonstrate the limitations of batching and explain the intuition behind `Chisel`.

In Fig. 2, we summarize the average request latency at low loads (minimal client overhead) when different number of keys are requested. The latency is decomposed into time spent at the client and at the memcached server, which we further divide into processing time and network time. Fig. 2 clearly shows that both the processing time at memcached and the network time increase linearly with the number of keys requested: (i) the processing time is composed of a constant time to process the memcached protocol headers and a variable time to fetch each key from memcached, which grows linearly with the number of keys requested; (ii) the network time can be seen as a constant time of RTT and a variable time to transfer the values on the network, which also grows linearly with the number of keys requested.

Mice Requests. Batching all keys in a single query helps to share the constant time (processing the protocol headers at memcached and network RTT), compared to retrieving every key in individual memcached queries. For example, a mouse request with 4 keys when retrieved in a single query takes about 119 μs of memcached time but when retrieved in 4 sequential queries takes 376 μs. An alternative is to query all the 4 keys in parallel (splitting), which takes about 95 μs. Intuitively, splitting helps amortize the variable time (fetching the keys from memcached and transferring the values over the network) as these steps are done in parallel. In this case the gains from splitting are not high because the number of keys requested is small. In addition, gains from splitting also depend on other constraints such as available threads at the client, the server, and the workload intensity.

Elephant Requests. Next, let us look at elephant requests with 100 keys. Batching all keys into one memcached query takes 880 μs, whereas *splitting* it into multiple parallel queries can take between 94 to 9400 μs depending on the degree of parallelism. In contrast to mice requests, for elephant requests the constant time of processing the protocol headers and network RTT (86 μs) is significantly smaller than the variable time of fetching keys from memcached and transferring values over the network (780 μs). When the keys requested are large, batching all keys into one query easily misses out the significant gains from parallel execution. However, splitting is not a panacea as the high fanout of perfect parallelism (as many queries as keys) increases the risk of sequential query execution due to the unavailability of idle threads. Under high load, threads become scarce and splitting may backfire unless the degree of parallelism is carefully tuned. We discuss this trade-off in section III-C.

Discussion. The observation that splitting large keys cuts the latency significantly is key to `Chisel`'s approach in handling skewed requests. As shown in Fig. 1, under a skewed workload of mice and elephant requests, when requests are only batched, mice requests experience high latency as they end up waiting behind elephant requests that take much longer to respond. `Chisel` aims to cut this waiting time by splitting the elephant requests in parallel. Adequate splitting also slashes down

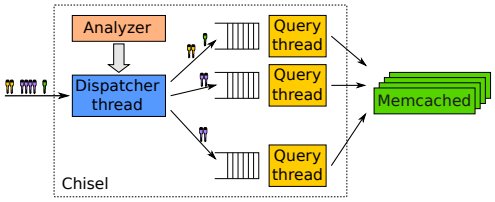


Fig. 3: Illustration of the `Chisel` architecture.

the latency of elephant requests as they exploit available parallel resources. In addition, `Chisel` merges multiple mouse requests to pool overheads and amortize queuing times under high loads (discussed in section III-B). `Chisel` appropriately tunes the split and merge levels, adapting to workload conditions that make either or both strategies more effective to cut latency.

III. Chisel

`Chisel` is a client-side solution implemented on top of a traditional memcached library to re-shape the request sizes by applying two main operations: split and merge. On the one hand, `Chisel` *splits* large user requests, made up of a large number of keys, into multiple sub-requests, to exploit the memcached parallelism. On the other hand, `Chisel` *merges* multiple small user requests into a single large query to amortize the query processing overhead across many small requests. The challenge however lies in deciding the right split and merge levels. This is especially difficult since, as we see next, their impact on the user-perceived latency depends on a number of factors, such as the workload, the actual request sizes, and the amount of available parallelism on both the client and the server.

We first explain `Chisel`'s architectural design and implementation details. Second, we illustrate the key workload and system parameters that affect the decisions of split and merge.

A. Architecture and Implementation

`Chisel` consists of three main components: (i) an analyzer, (ii) a dispatcher thread, and (iii) a pool of query threads (see Fig. 3). The *analyzer* determines the split and merge levels, i.e., the number of chunks to split elephant requests and the number of mice requests to merge into a single request. Due to the complexity involved in choosing the split and merge levels, the core of the analyzer is a set of stochastic models that predict the expected latency under different split and merge configurations (see Section IV). The *dispatcher* and *query* threads actuate the optimal split and merge decisions determined by the analyzer.

Dispatcher Thread Upon receiving a user request, the dispatcher thread checks the request size (number of requested keys) and takes the following actions accordingly. For elephant requests, the dispatcher thread splits them into chunks or sub-requests according to the optimal number of chunks determined by the analyzer, and dispatches these subrequests to the queues of different query threads to enable parallel retrieval. The queues are chosen in a round robin fashion. For mice requests, the dispatcher appends the optimal number of requests to merge (B^*) as metadata to the request and sends it to the query threads. After forwarding B^* consecutive mice requests to one query thread, the dispatcher selects a new query thread in round robin order to process the next B^* mice requests as one merged query. Let us take $B^* = 3$ as an example. In this case the dispatcher sends three consecutive mice requests to the same query thread queue before moving on to the next thread in a round robin fashion. We thus quickly accumulate B^* mice requests to merge at the same query thread, avoiding unnecessary delays. To allow this optimization without the need of synchronization, and since the overhead of processing a user request by the dispatcher is low, we opted to have a single dispatcher thread.

Query Threads All (sub)requests wait at the queues in front of query threads, which in turn process (sub)requests in a first-come-first-served fashion. For every (sub)request at the head of queue, the query thread first checks the optimal number of requests to merge saved in its metadata while setting up the communication with one of the memcached servers. The overhead imposed at the query thread depends on the library implementation and additional functionalities required, e.g., consistency checks. The memcached protocol allows to retrieve keys via two commands: one key at a time (`get` command) or multiple keys together (`gets` command) [13]. As the (sub)requests in the query threads' queues have a list of requested keys, all query threads use the second alternative only.

Query threads try to piggyback (sub)requests according to the optimal number of requests to merge, under the condition that subsequent requests arrive before completing the connection to the memcached servers. In other words, the setup time is the upper limit on the additional wait to merge multiple (sub)requests. This limit also bounds the search for the optimal merging level performed with the latency model implemented by the analyzer (see Section IV). Another critical parameter here is the number of query threads. `Chisel` sets the number of query threads at least as high as the available memcached threads, so as to better leverage the available resources. In case the memcached servers are faster to return query results than query threads to process (sub)requests, it is possible to set a higher number of query threads than memcached threads. The impact of such choice is also reflected in the latency models implemented by the analyzer. We note that the analyzer can set the optimal number of requests to merge and the

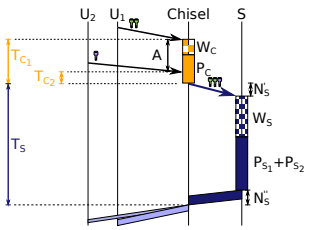


Fig. 4: Example of merging mice requests.

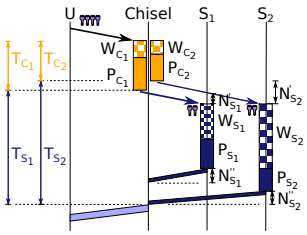


Fig. 5: Example of splitting elephant requests.

optimal number of chunks to one. In such case, all keys in a request are processed in a single memcached query as performed in state-of-the-art memcached clients. This mode of operation is the same as *batching*.

System Assumptions We particularly focus on the scenario of read-dominated workloads in a single replica pool, where there are only a small number of memcached servers. Consistency within the replica set is ensured by broadcasting updates across all its members. We do not assume that all the data stored is available in one replica set, instead each `Chisel` client focuses on traffic directed to a single replica pool.

B. Merging Mice Requests

First, we distinguish the difference between batching and merging, and then highlight the conditions under which merging improves latency.

Batching vs. Merging. While batching aggregates multiple keys of a single user request into one memcached query, merging aggregates multiple user requests into a single memcached query. It defers the execution of a user request by waiting to group multiple user requests into one batch, which is then forwarded as a single query to Memcached. Every user, then, experiences a latency greater than or equal to the time it takes to process all merged users requests. At the outset merging seems to introduce additional delays in exchange for little to no benefits. Actually, we show that merging multiple users requests reduces the latency under specific circumstances, particularly when any client overhead can be pooled together or when requests queue at the client.

Overhead Pooling and Queuing. The overhead at a client can range from connection handling to data and consistency management. The benefits of merging user requests begin to manifest when the client overhead can be pooled together and executed in parallel with the waiting time for merging user requests. In Fig. 4, we illustrate the merge operation via an example of merging two requests. User 1 sends a request for 2 keys which are then immediately forwarded by the dispatcher thread to a query thread. While the query thread sets up the connection with one of the memcached servers, another mouse request from User 2 is forwarded to the same query thread by the dispatcher thread. The query thread piggybacks request 2 with request 1 and sends them as a single query. The client time (T_C) time for request 2 reduces and the setup time is amortized over the two requests. Moreover, as the two requests are processed in the same memcached query, they have the same request submission time (N'_S), waiting time (W_S), processing time ($P_{S_1} + P_{S_2}$), and value transfer time ($N''_{S_1} + N''_{S_2}$).

Our experimental results (omitted due to lack of space) indicate that at high loads merging multiple mice requests cuts latency even in the absence of any overhead.

C. Splitting Elephant Requests

To mitigate the performance degradation caused by elephant requests, `Chisel` proactively splits these requests into a set of smaller memcached queries or chunks. The optimal number of chunks to split a request depends on the load at client and servers, the thread availability, as well as the overhead of splitting, which needs to be highly optimized to fulfill the sub-millisecond latency requirements of in-memory data stores such as memcached.

We illustrate the gain and overhead of splitting with the example in Fig. 5, where an elephant request is split into two memcached queries or chunks. As soon as the request arrives at `Chisel`, the dispatcher thread splits the elephant request into two subrequests and enqueues them to two different query thread queues in a round-robin fashion. The subrequests wait until they reach the head of the queue (W_C) and are then processed and forwarded by the query threads (P_C) to the memcached servers. We refer to the sum of these two times as the `Chisel` client time T_C[3]. Once forwarded, each query is sent over the network (N'_S), waits at the memcached server (W_S), is processed to retrieve the keys (P_S), and the resulting values are transferred back to the query thread (N''_S). All these times together make up the memcached server time T_S. Each subrequest experiences different T_C and T_S times and the user-perceived latency is determined by the last subrequest that completes. The gains from splitting thus come from parallelizing the server time (T_S).

In our experimental results under high load the client time increases with the number of chunks due to the

[3]We denote in general the waiting, processing and network and total time with the letters W, P, N and T, respectively, where $N = N' + N''$, and use the subscripts C and S to denote the client and server.

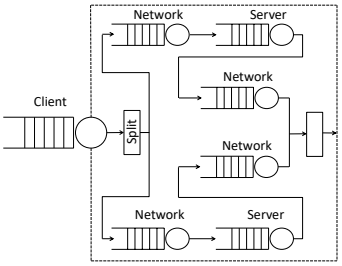


Fig. 6: Chisel's Layered model: client service times include response times at server and network.

splitting overhead and the time waiting for available query threads. In contrast, with low load Chisel is able to better harvest the parallelism at the client and the memcached servers using the split operation.

D. Split Plus Merge

The previous sections illustrate that, for either splitting or merging, the load intensity is critical to determine the trade-off between the performance gain and the additional overhead at both client and memcached servers. At very low loads, it is beneficial to split elephant requests into multiple chunks and not to merge any mice requests. On the other extreme, at very high loads, it is beneficial to merge multiple mice requests and not split elephant requests. For loads in between, different combinations of merging and splitting provide different benefits. It is extremely difficult to learn the optimal split and merge levels via offline profiling as the waiting times at the query queues and within the memcached servers depend on the complex interplay of workload dynamics, system configurations, and split/merge operations themselves. In addition, as Chisel performs both split and merge concurrently on different sets of requests (mice and elephants), their combined impact significantly increases the difficulty to obtain the optimal split and merge levels. We thus develop stochastic models to support the core operations of the analyzer: (i) to estimate the queueing times when simultaneously applying the split and merge operations, and (ii) to determine the optimal split and merge levels. Moreover, unlike the optimal splitting level the model parameters can be easily profiled under low loads.

IV. Analyzer Main Features

It is challenging to decide upon the *right* split and merge levels. To tackle this complexity we develop a set of stochastic models that obtain the expected user-perceived latency for a given configuration. The Chisel analyzer implements these models to find split and merge levels that result in the lowest predicted latency.

1. Layered Operation. The first key aspect considered in the model is that the request processing times at the Chisel client incorporate the delays at other resources, namely the network and the memcached server. Figure 6 illustrates this layered dependency via a high level queueing model for the splitting operation. To process a request the Chisel client first splits it in chunks and forwards them to different servers. Each chunk then experiences subsequent steps to be completed, i.e., network delay, memcached server time, and network delay again. During these steps, the Chisel thread handling the user request remains busy. This layered dependency of the client thread holds for both split and merge operations, as shown in their system illustrations of Figures 4 and 5. To capture such a dependency, we use a *layered* model, where the service times at one layer are made of the response times at other layers. We thus first analyze the memcached server and network layers to obtain their latency, and employ the results to analyze the Chisel client layer and obtain the request total latency.

2. Explicit Overhead in Query Processing and Network times. Our model captures the amortization of processing and network overheads in the split and merge operations. As illustrated in Section II-B, both the query processing times and the network transfer time are composed of a constant time and a variable time. The examples presented in Section II-B clearly illustrate that the specific values of these constant times can be very significant when deciding the appropriate split and merge levels. The proposed models *explicitly* consider that request processing time is made of constant and variable times, and that the latter depends on the number of keys in a query. The models are thus able to capture the impact that split and merge have on the request latency. Furthermore, the models incorporate both the server and network overhead, where the total transfer time is also made of a constant and a variable time.

3. Capturing the Impact of split and merge on Latency. The proposed models also take into account that both split and merge generate additional overheads. Consider for instance the case of splitting shown in Figure 6. To put together the user response the Chisel client must wait for the results of *all* chunks to arrive, generating a delay that would not be present in the case without splitting. This requires approximating a fork-join-like behavior by means of harmonic numbers together with fairly general processing and inter-arrival times [14]. In the case of merging, the models incorporate the additional delay caused when accumulating several requests into a single one. Further, combining elephant and mouse requests requires considering multi-class traffic, which we capture as a marked markovian arrival process [15].

4. Approximate Analysis for Quick Scenario Evaluation. Given the sub-millisecond operation of memcached data stores, the proposed models provide *approx-*

imate closed formulas for the overall request latency, which can be quickly evaluated for a number of Chisel configurations. Evaluations in Section V show that the models are in fact accurate and can thus support the selection of the optimal split and merge levels.

All in all, the proposed models allow us to quickly evaluate the expected latency for a given Chisel configuration. The main knobs to consider are the number of mice requests to merge B and the size of the chunks in which the elephant requests are split L. The Chisel analyzer exploits the model to evaluate many possible combinations of B and L via a grid search and chooses the one that provides the smallest expected request latency. The experiments presented in the next section show that the model is able to provide accurate results to support the selection of the right split (L) and merge (B) levels.

V. Evaluation

In this section we extensively evaluate Chisel as a mechanism to improve latency under various request size mixes and load conditions. Particularly, we deploy Chisel on memcached hosted on the testbed described in Section V-A. We focus on the mean and 95^{th} percentile of user perceived latency as main performance metrics. We show that Chisel can achieve significant latency gains against the state-of-practice of retrieving all keys of one request in a single memcached query, i.e., using only batching.

A. Setup and Scenarios

Testbed. The testbed is composed of identical physical servers, used to either host memcached or Chisel collocated with an in-house memcached load generator. Each server is equipped with two Intel Xeon E5-2630v3 CPUs, 128 GB DDR4 RAM, six 1-TB solid state disks in RAID5, and dual 10-Gigabit Ethernet adapters with Jumbo frames enabled. Servers are connected in a star topology via a Cisco 9500 switch. We use memcached server v1.4.31 configured with 4 threads to avoid thread scaling issues [16].

Chisel. We implement Chisel in the go programming language as a layer on top of the gomemcache library handling the low level memcached protocol. We use the golang v1.7.3 compiler as we noticed significant performance improvements with respect to older versions. The binary combines Chisel with an in-house load generator with support for different arrival and request size distributions. We also implement a highly efficient version of Chisel in C, using libevent, that is capable of achieving throughputs of around 800,000 requests per second.

Scenarios. We generate mixed user requests, whose inter-arrival times (A) follow an exponential distribution with mean $\{50, 60, 70, 90, 120, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100\}\mu$s. We consider three sizes for mice request, i.e., 1, 5, and 10 keys, while elephant requests ask for 100 keys in all scenarios. In terms of mice-to-elephant request mixes, we consider 9 to 1 and 5 to 1 ratios, i.e., for every elephant request we inject on average 9 or 5 mice requests. We emulate different memcached client overheads by artificially injecting waiting times of roughly 0, 60, 120, and 180 μs.

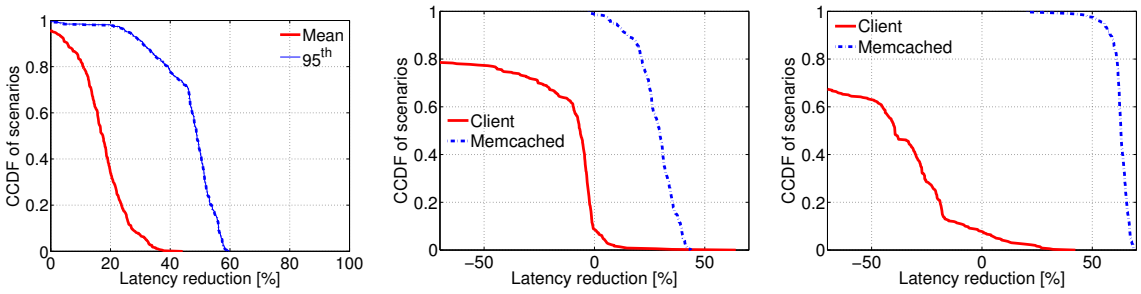
Baseline. As baseline key retrieval policy we use batching-only, the state-of-practice where all key-value pairs in a user request are processed in a single memcached query by our Chisel client, i.e., both splitting and merging are disabled. All requests are processed in parallel by the query threads, i.e. the baseline is an optimized solution, rather than a naive approach.

Metrics. Due to the large number of combinations of the aforementioned parameters and the space limit, we mainly show aggregate results, i.e., the average and standard deviation from multiple scenarios. We particularly present the performance gain on the mean and 95^{th} percentile latency, i.e., the percentage of latency reduction from the baseline case. The higher the latency gain, the better Chisel performs. We refer to the 95^{th} percentile latency as tail latency here on.

Profiling of P_S and N_S. The analyzer requires separate models of the processing P_S (processing time at memcached) and network N_S (time to transfer values from memcached to Chisel) times for the memcached queries across different request sizes. We obtain these via offline profiling on the testbed using low load runs with different fixed request sizes. During the offline profiling, we instrument memcached to report P_S. We collect the memcached time T_S from Chisel, and estimate N_S as the difference $T_S - P_S$. We find that P_S and N_S are approximated by the following two linear equations: $P_S = 3.3 + 0.6 * K$ μs and $N_S = 85.0 + 6.5 * K$ μs, where K is the number of keys.

B. Chisel Aggregate Results

We first evaluate a single Chisel over 360 scenarios. Fig. 7 summarizes the results. Fig. 7(a) shows the complementary CDF (CCDF) of the performance gains on mean and tail latency. The higher tail latency curve (compared to the mean latency curve) immediately indicates that tail latency reductions achieved by Chisel are better than for the mean. On average, Chisel is able to reduce mean latency by 16% and 95^{th} percentile by 46.5% compared to the baseline policy. As Chisel proactively splits elephant requests - a root cause of high tail latency, Chisel can effectively reduce the tail compared to the baseline. Zooming into the CCDF of the tail latency, we also see that Chisel achieves at least



(a)Mean and 95%ile gains (b) Gain breakdown - mean (c) Gain breakdown - 95%ile

Fig. 7: Bare-metal: CCDF of overall performance gain, gain on the client and on memcached, and optimality ratio over 360 scenarios.

a 50% improvement for more than half of the scenarios considered. The low latency gains of tail mainly happen at the scenarios with high loads, i.e., small inter-arrival times, e.g., $A = 50$, 60 and 70 μs. Moreover, the sharp fall of the CCDF indicates that performance gains are similar across many scenarios, supporting the robustness and adaptability of `Chisel` to different operation points.

Considering the lower latency reduction for the mean, we observe that the request size mixes evaluated here are dominated by mice requests, which are improved by the merging operation during high loads. This corresponds to a small number of scenarios with low inter-arrival rates, e.g., $A = 50$ and 60 μs. Also, roughly 40% of the scenarios have mean latency reductions greater than 20%, which mainly correspond to scenarios with a 5-1 mice to elephant mix, with mice request size of one key, and under lower loads where elephants are better split.

We further break down the performance gains by the client and memcached times in Fig. 7(b) and (c), comparing them with the client and memcached times of the baseline. On the one hand, the client time of `Chisel` actually increases compared to the baseline in roughly 90% of scenarios, for both the mean and tail latency. On the other hand, the memcached time is reduced in almost all scenarios, resulting in an average reduction of 30% and 70% for mean and tail latency, respectively. `Chisel` essentially improves the memcached time at the cost of a higher client time, especially due to the overhead of the additional `Chisel` split and merge operations.

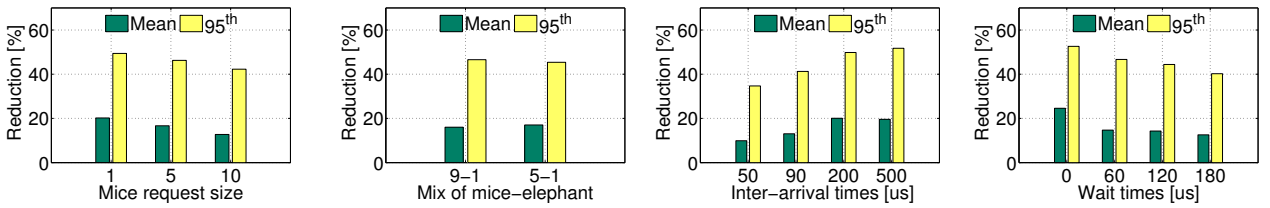
C. Sensitivity Analysis

Here we present a detailed sensitivity analysis to show the impact of workload mixes, mice request sizes, loads, and client overheads. To this end we compute the average latency reduction for each condition of interest, see Fig. 8. When the size difference between mice and elephant requests is bigger, e.g., 1 v.s. 10 keys per mice request, `Chisel` achieves higher latency reductions by proactively resizing memcached queries to reduce their variability. Consequently, Fig. 8(a) shows a slightly decreasing trend in gain as mice request size increases. Fig. 8(b) depicts that there is no difference in tail latency reduction between 9-1 and 5-1 mixes, whereas reduction in mean slightly increases. This is due to `Chisel` being able to split the more frequent elephant requests under the 5-1 mix, increasing the benefit for mice requests as their probability of waiting behind big elephants reduces.

Fig. 8(c) shows how `Chisel` performs against a set of selected inter-arrival times. With increasing A, the reduction in mean and tail latency increases. Under high load, `Chisel` tends to split elephant requests into fewer chunks due to the reduced degree of parallelism. Consequently, query variability is less mitigated under low inter-arrival times. In contrast, merging requests is better suited during high loads, although its effectiveness is less prominent than the one of split operation. Regarding client overhead, Fig. 8(d) shows that the impact of a larger overhead is similar to lowering inter-arrival times. The reduction in mean drops significantly from no-overhead to 60 μs overhead and then stabilizes, whereas the tail reduction shows a rather linear trend with respect to the overhead considered. The increasing overhead essentially makes the overall constant term per user request more dominant, weakening the benefit of using parallelism to reduce the variable term. In summary, `Chisel` is particularly effective for skewed request sizes and light-weight client implementations.

D. Effective Query Sizes

Here we present the average effective query sizes to support our argument that one of the key reasons for the gains achieved by `Chisel` is its ability to reshape the memcached query sizes. Fig. 9 summarizes the average size before and after applying `Chisel` in a high load ($A = 120\mu$s) and a low load ($A = 1000\mu$s) scenario. We show six different cases made up of two request size mixes and three different mice request sizes. We note that these two load levels also represent the minimum and maximum reduction in average memcached query sizes.



(a) Key sizes of mice requests (b) Mixes of requests (c) Inter-arrival times (d) Injected client overhead

Fig. 8: The latency gain of Chisel, under different conditions. Each bar presents the average value for multiple scenarios where the specified conditions hold true.

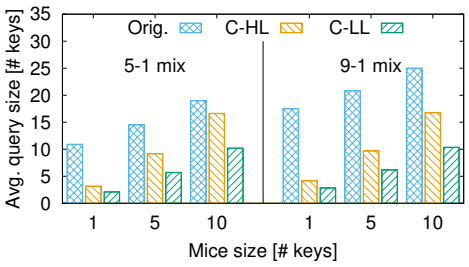


Fig. 9: Average query size: original, Chisel at high load (C-HL), and low load (C-LL) across mice asking 1,5,10 keys and 5-1, 9-1 mice-elephant mixes.

Chisel is able to reduce the average size significantly compared to the baseline, particularly under low load due to the splitting operation. Comparing the mixes of 9-1 and 5-1, the difference between the baseline and Chisel-LL is higher for 5-1 for all the mice sizes considered. This is because Chisel has more chances to apply splitting on the elephant requests. The largest reduction in average query size with respect to the baseline is with 1-key mice requests and a 5-1 mix.

VI. Related Work

As the core design principle of Chisel is to adaptively change the query size to retrieve users' requests for in-memory stores to optimize the latency, we structure the prior art in two main areas.

Latency optimization for in-memory data stores. Due to the effectiveness of in-memory stores to scale the performance of large-scale social network services [17, 18], significant efforts from industry and academia try to optimize their performance, either by directly modifying the memcached implementation [4, 6] or by altering the request scheduling [7, 5]. Social network companies, such as Facebook [1] and Twitter [2], not only make their memcached implementation available on public repositories but also include their proxy servers, e.g., MCrouter [19], which load balance queries across a large number of memcached servers. Novel techniques on hashing, cache replacement [4], and data structures [6] enable efficient and parallel access to a single memcached server. To improve the performance of the entire cluster, various distributed and centralized load balancing techniques are proposed to mitigate the tail latency degradation, relying on consistent hashing [20], key popularity [7, 21], and key replications [22, 23].

While the focus of these works is on the latency spent at the memcached servers only, our interest is on the user perceived latency, a super set of memcached time and client time. Chisel is compatible with state of the art memcached implementations as well as ready to be integrated with existing load balancing policies. Moreover, Chisel uses the novel idea of splitting (merging) user requests into multiple (a single) query, which has not been explored before in the context of in-memory stores.

Modeling and Managing Skewed Sizes. There is a number of analytical studies modeling the impact of executing skewed workloads with big and small jobs and proposing novel scheduling policies [24, 25, 26] to improve particularly the tail latency. The presence of big jobs [27, 28] can significantly hurt the small jobs that wait behind. One particular challenge to address with mixed workloads is the lack of size information prior to execution. Shortest remaining processing time [29] has been shown to be a near optimal scheduling policy, even with inexact job size information. Rein [3] propose a multi-get aware scheduling to cut tail latency.

However, most analyses do not consider the option of altering the job sizes as the workloads are assumed to be uncontrollable. Instead, we derive a set of layered queueing models that can approximately capture the request latency under skewed workloads and employ the novel query sizing strategy proposed in Chisel. Moreover, Chisel can also adopt existing scheduling policies to further improve the latency.

VII. Concluding Remarks

In this paper we present a novel solution, Chisel, which can effectively improve the mean and tail request latency in data stores where requests display a skewed number of key-value pairs. Chisel splits elephant requests into parallel queries while merging mice requests into single queries, adaptively shaping the queries' sizes and level of parallelism guided by novel

layered queueing models. `Chisel` is able to achieve nearly optimal latency results from merging and splitting requests as evidenced by our empirical evaluation. We extensively evaluate `Chisel` on a testbed using two prototype implementations. Overall, the parallel design and implementation of `Chisel` is able to improve both the mean and tail latency, over hundreds of scenarios with different request arrivals and size distributions. For our future work, we intend to validate the optimality of the model-based decisions and further extend the evaluation to different system platforms and multi-client scenarios.

VIII. Acknowledgements

This research was partly funded by the SNSF NRP75 project Dapprox 407540_167266.

References

[1] R. Nishtala, H. Fugal *et al.*, "Scaling memcache at Facebook," in *NSDI*, 2013.

[2] "Twemcache," https://github.com/twitter/twemcache.

[3] W. Reda, M. Canini *et al.*, "Rein: Taming tail latency in key-value stores via multiget scheduling," in *EUROSYS*, 2017.

[4] B. Fan, D. G. Andersen *et al.*, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *NSDI*, 2013.

[5] B. Fan, H. Lim *et al.*, "Small cache, big effect: provable load balancing for randomly partitioned cluster services," in *SoCC*, 2011.

[6] H. Lim, D. Han *et al.*, "MICA: A holistic approach to fast in-memory key-value storage," in *NSDI*, 2014.

[7] W. Zhang, T. Wood *et al.*, "Netkv: Scalable, self-managing, load balancing as a network function," in *IEEE ICAC*, 2016.

[8] J. F. Pérez, R. Birke *et al.*, "Dual scaling vms and queries: Cost-effective latency curtailment," in *ICDCS*, 2017, pp. 988–998.

[9] H. Sun, R. Birke *et al.*, "Accstream: Accuracy-aware overload management for stream processing systems," in *ICAC*, 2017, pp. 39–48.

[10] R. Birke, M. Björkqvist *et al.*, "Meeting latency target in transient burst: A case on spark streaming," in *IC2E*, 2017, pp. 149–158.

[11] M. Jeon, Y. He *et al.*, "TPC: target-driven parallelism combining prediction and correction to reduce tail latency in interactive services," in *ASPLOS*, 2016.

[12] "Asynq," https://engineering.quora.com/Asynchronous-Programming-in-Python.

[13] "Memcached," https://memcached.org/.

[14] G. Bolch, S. Greiner *et al.*, *Queueing Networks and Markov Chains*. Wiley, 2006.

[15] G. Latouche and V. Ramaswami, *Introduction to matrix analytic methods in stochastic modeling*. SIAM, 1999.

[16] N. Gunther, S. Subramanyam *et al.*, "Hidden scalability gotchas in memcached and friends," in *VELOCITY*, 2010.

[17] Y. Xu, E. Frachtenberg *et al.*, "Characterizing facebook's memcached workload," *IEEE Internet Computing*, vol. 18, no. 2, pp. 41–49, 2014.

[18] B. Atikoglu, Y. Xu *et al.*, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012.

[19] "MCrouter," https://github.com/facebook/mcrouter.

[20] D. G. Andersen, J. Franklin *et al.*, "FAWN: a fast array of wimpy nodes," *Commun. ACM*, vol. 54, no. 7, pp. 101–109, 2011.

[21] Y. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *SoCC*, 2013.

[22] Y. Cheng, A. Gupta *et al.*, "An in-memory object caching framework with adaptive load balancing," in *EUROSYS*, 2015.

[23] V. Jaiman, S. B. Mokhtar *et al.*, "Héron: Taming tail latencies in key-value stores under heterogeneous workloads," in *37th IEEE Symposium on Reliable Distributed Systems, SRDS 2018, Salvador, Brazil, October 2-5, 2018*, 2018, pp. 191–200.

[24] M. Harchol-Balter, B. Schroeder *et al.*, "Size-based scheduling to improve web performance," *ACM Trans. Comput. Syst.*, vol. 21, no. 2, pp. 207–233, 2003.

[25] J. Nair, A. Wierman *et al.*, "Tail-robust scheduling via limited processor sharing," *Perform. Eval.*, vol. 67, no. 11, pp. 978–995, 2010.

[26] B. Schroeder, M. Harchol-Balter *et al.*, "How to determine a good multi-programming level for external scheduling," in *ICDE*, 2006.

[27] M. Harchol-Balter, K. Sigman *et al.*, "Understanding the slowdown of large jobs in an M/GI/1 system," *SIGMETRICS Perf. Eval. Rev.*, vol. 30, no. 3, pp. 9–11, 2002.

[28] S. Spicuglia, M. Björkqvist *et al.*, "On load balancing: a mix-aware algorithm for heterogeneous systems," in *ACM/SPEC ICPE*, 2013, pp. 71–76.

[29] A. Wierman and M. Nuyens, "Scheduling despite inexact job-size information," in *SIGMETRICS*, 2008.