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

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

Summarized by: John Murray
INTRODUCTION:

What is a Key-Value store?

What is Tail Latency?

What is Chisel?

What is an Analyzer thread?

What is a Dispatcher thread?

What is a Query thread pool?

What is Memcached?
**KEY-VALUE STORE DB**

Uses an associative array (map/dictionary) where each key is associated with one & only one value in a collection (1:1 relationship).
Four sources of packet delay

\[ d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}} \]
TAIL LATENCY

• While less latency is always better, average latency is still OK because it’s predictable
• ‘Average’ latency is the devil you know
• Tail latency, however, is the type of latency spikes that will make you serve dry turkey to your grandma
• Website loading comes out uneven resulting in very bumpy data transfer
• It’s not predictable & you can’t plan for it

http://blog.virtualstoragezone.com/what-the-heck-is-tail-latency-anyways/
MEMCACHED

High-performance, distributed memory data store & caching system used to speed up dynamic database-driven websites by reducing database reads
ANALYZER, DISPATCHER, QUERY THREADS:
Background & Motivation:

• Size of user requests has been shown to be highly skewed in production systems
• Skewed workloads degrade performance
• Batching reduces overall latency through aggregation of many requests to single query
• Batching suffers disadvantages in processing skewed requests
Chisel’s Main Contributions:

- Has the ability to *reshape* query sizes by intelligently combining split & merge operations.
- Takes a **holistic** approach to both latency & parallelism.
- **Adaptable** to varying workloads, system conditions, adjusting split & merge levels.
Chisel Architecture:

a) Analyzer: determines split & merge levels between number of elephant & mice requests to form a single request

b) Dispatcher Thread: checks size (# keys) of user request splitting elephant requests into chunks while merging mice requests together

c) Query Threads: piggyback sub-requests according to the optimal number of requests required to merge before completing (faster) Memcached server connections
Fig. 4: Example of merging mice requests.

Fig. 5: Example of splitting elephant requests.
# Mice Merging vs. Batching

<table>
<thead>
<tr>
<th>Merging aggregates multiple user requests into a single Memcached query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batching aggregates multiple keys of a single user request into one memcached query</td>
</tr>
<tr>
<td>Merging multiple user requests reduces latency most effectively when overhead can be pooled or queued together</td>
</tr>
</tbody>
</table>
SPLITTING ELEPHANT REQUESTS:

- Elephant requests are **proactively split** into a set of smaller Memcached queries to **mitigate performance degradation**

- The optimal number of chunks to split a request into is dependent upon:
  - a) Current load at client & servers
  - b) Thread availability
  - c) Overhead of splitting

- Under **high load** client time increases with # of chunks due to splitting overhead & time waiting for available threads

- Under **low load** Chisel is better at harvesting the **parallelism** at the client & Memcached servers using split operation
SPLIT PLUS MERGE:

• At **low loads** it is **optimal** to split **elephant requests** into multiple chunks **without merging mice requests**

• At **high loads** it is **optimal** to merge multiple **mice requests without splitting elephant requests**

• For **middle loads**, it is tricky to learn the optimal split & merge levels

• Therefore **Stochastic methods were developed** for the Chisel analyzer to:
  • **Estimate queueing times** when simultaneously applying the split & merge operations
  • Determine the optimal split & merge levels
ANALYZER MAIN FEATURES:

- Layered Operation finds split & merge levels with the lowest predicted latency
- Explicit Overhead in Query Processing & Network times captures amortization overhead of split & merge operations
- Capturing the Impact of split & merge on Latency combining elephant & mouse requests requires considering multi-class traffic
- Approximate Analysis for Quick Scenario Evaluation provide closed formulas for overall request latency
• Setup & Scenarios
  • Chisel is implemented in the GO programming language handling the low-level Memcached protocol with mixed user requests, different Memcached client overheads & artificially injected waiting times

• Chisel Aggregate Results
  • Higher tail latency curve immediately indicates that tail latency reductions achieved by Chisel reduce mean latency by 16% by proactively splitting elephant requests (root cause of high tail latency)

• Sensitivity Analysis
  • Under high load Chisel splits elephant requests into fewer chunks due to reduced degree of parallelism
  • Conversely, merging requests is better suited during high loads with effectiveness less prominent than with the split operation

• Effective Query Sizes
  • Chisel reduces average query size significantly compared to the baseline under low load due to the split operation as there are more chances to apply splitting to elephant requests
RELATED WORK:

• Latency Optimization for in-memory data stores
  • Chisel is compatible with the latest Memcached implementations & integrates with existing load balancing policies
  • Chisel uses splitting/merging user requests into multiple/single queries which has never previously explored within the context of in-memory stores

• Modeling & Managing Skewed Sizes
  • Shortest remaining processing time has been shown to be a near optimal scheduling policy
  • Chisel can adopt existing scheduling policies to even further improve latency dynamically
WRAP-UP!

• Chisel effectively improves the mean & tail request latency within Memcached data stores where requests display a skewed number of key-value pairs

• Chisel splits elephant requests into single queries, adaptively shaping the query sizes & level of parallelism

• Chisel is able to achieve near-optimal latency results from both merging & splitting requests

• Through it’s parallel design Chisel is able to improve the mean & tail latency of over hundreds of scenarios of varying request arrival & size distributions