Adaptive Performance-Aware Distributed Memory Caching

Jinho Hwang and Timothy Wood
George Washington University
Background: Memory Caching

- Two orders of magnitude more reads than writes
- Solution: Deploy memcached hosts to handle the read capacity

![Diagram showing the process of memory caching]

1. HTTP Request
2. Get(key)
3. Miss(key)
4. DB Lookup(key)
5. Data
6. HTTP Response
6. (key, data)
Memcached at Scale

- Databases are hard to scale… Memcached is easy
  - Facebook has 10,000+ memcached servers

- Partition data and divide key space among all nodes
  - Simple data model. Stupid nodes.

- Web application must track where each object is stored
  - Or use a proxy like moxi
Scales easily, but loads are imbalanced

- Random placement…
- Skewed popularity distributions…

Load on Wikipedia’s memcached servers
Motivation

- Consistent hashing does not evenly load data across memory cache servers
  - Variation in number of keys assigned to each server
  - Key popularity is skewed and changes over time

- Solution: dynamically balance load according to the performance

Based on Wikipedia 2008 database dump and access trace
Contributions

- A hash space allocation scheme
  - allows for targeted load shifting between unbalanced servers
- Adaptive partitioning of the cache’s hash space
  - automatically meet hit rate and server utilization goals
- An automated replica management system
  - adds or removes cache replicas based on overall cache performance
Outline

- Background and Motivation
- Initial Hash Space Partitioning
- Dynamic Adaptation
- Evaluation
- Conclusions
Background: Hash Space Allocation

- **Simple Hashing**
  - hash(key) % [# of server]
  - Once assigned, never changes
  - If node added or removed, all objects need to be rearranged

- **Consistent hashing**
  - Treat hash space as ring with nodes assigned to each region
  - Node addition / removal only affects adjacent nodes
  - Used in P2P systems and by popular memcached proxy system Moxi
To enable efficient repartitioning of the hash space:
- Every node is adjacent to every other node
- This allows a simple transfer of load between two nodes by adjusting just one boundary

Required number of duplicate nodes = $v \geq \frac{n_0 P_2}{n_0} = n_0 - 1$, 
Total number of nodes = $n_0 \times (n_0 - 1)$
Multiply number of virtual nodes
Dynamic Hash Space Scheduling

- Two factors to measure server performance:
  - Hit rate: enough memory for popular data
  - Usage ratio: server processing
- Minimize \( \text{cost} = \text{hit rate} + \text{usage ratio} \)

- Scheduling decision: \( \alpha \in [0, 1] \)
  - Find the most different two memory servers
  - Find the most different two adjacent virtual nodes

- Size of hash space moved at each scheduling decision
  - Determine the speed of adaptability, but more fluctuation
  - Using ratio value: \( \beta \in (0, 1] \)
Node Addition / Removal

- Balance out the requests across replicas that overall performance improves
- Highly overloaded server(s) sustaining a certain period of time should be backed by new server(s)
- Find the most costly memory server, and its virtual node
- Find the least costly memory server, and its virtual node
Outline

- Background and Motivation
- Initial Hash Space Partitioning
- Dynamic Adaptation
- Evaluation
- Conclusions
Experimental Setup

- Lab setup
  - Five experimental servers (4 × Intel Xeon X3450 2.67GHz processor, 16GB, and a 500GB 7200RPM hard drive)

- Amazon setup
  - 15 medium instances

- All workloads are from Wikipedia data and access traces
Initial Hash Space Assignment

- 5 memory servers used (total 500 virtual nodes)
  - For consistent hashing, 100 virtual nodes per each server
  - For our scheme, the initial set is $5 \times 4 = 20$, and 25 virtual nodes per node

- The largest gap between the biggest hash size and the smallest hash size is $381,114,554 \approx 20\%$ more
Dynamic Partitioning

- $\alpha = 1.0$ (only hit rate)

- $\alpha = 0$ (only usage ratio)
When $\alpha = 0.5$, $\beta = 0.01$

- Hit Rate
- # of Reqs per min ($\times 10^3$)
- Cost
- Hash Space ($0 - 2^{32}$)
Node Addition / Removal

- **Addition**
- A new node takes reduces load on the overloaded server

- **Removal**
- Removing an underloaded server gives cost benefits while maintaining performance

![Graph showing node addition and removal](image-url)
**β Behavior**

- Amount ratio of hash space movement
- Determine the speed of adaptability
- Use $\beta = 0.01$ (1%) to show the behavior

Traffic changes over 5 hours

Moved hash space per each scheduling
Scaling Up / Down

- Dynamically add / remove server(s) depending on amount of load intensity
- Watch each server for a period of time (5 min) to check high load sustainability
- To maximize variation, $\alpha = 1$ (hit rate only)
- 5 Wikipedia traffic generators used
QoE Improvement

- Wikipedia workload achieves better response time as hit rate increases (≈ 45% increase)
- But the number of servers used increases as well
- As recommendation, the combination of hit rate and usage rate ($\alpha = 0.5$) is a good administrative choice
Related Work

- [Stoica, ToN 03] Chord Peer-to-Peer architecture
- [Nishtala, NSDI 13] Scaling Memcached at Facebook
- [Zhu, HotCloud 12] Shrinking memcached to save $$
- Ideas may apply to many other key-value based storage systems: couchbase, redis, SILT, FAWN, etc
Conclusion

- **Summary**
  - A hash space allocation scheme
    Carefully place nodes to ensure adjacency
  - Adaptive partitioning of the cache’s hash space
    Maximize hit rate and minimize difference in utilization rate
  - An automated replica management system
    Detect sustained overload and add or remove nodes

- **Future works**
  - Automatic \( \alpha \) value adjustment to minimize response time
  - Targeted management of hot objects without impacting application performance