Mercury: Bringing efficiency to key-value stores

Rohan Gandhi
Purdue University
gandhir@purdue.edu

Aayush Gupta
IBM Research - Almaden
guptaaa@us.ibm.com

Anna Povzner
IBM Research - Almaden
apovzne@us.ibm.com

Wendy Belluomini
IBM Research - Almaden
wb1@us.ibm.com

Tim Kaldewey
IBM Research - Almaden
tkaldew@us.ibm.com

ABSTRACT

While the initial wave of in-memory key-value stores has been optimized for serving relatively fixed content to a very large number of users, an emerging class of enterprise-scale data analytics workloads focuses on capturing, analyzing, and reacting to data in real-time. At the same time, advances in network technologies are shifting the performance bottleneck from the network to the memory subsystem. To address these new trends, we present a bottom-up approach to building a high performance in-memory key-value store, Mercury, for both traditional, read-intensive as well as emerging workloads with high write-to-read ratio. Mercury's architecture is based on two key design principles: (i) economizing the number of DRAM accesses per operation, and (ii) reducing synchronization overheads. We implement these principles with a simple hash table with linked-list based chaining, and provide high concurrency with a fine-grained, cache-friendly locking scheme. On a commodity single-socket server with 12 cores, Mercury scales with number of cores and executes 14 times more queries/second than a popular hash-based key-value system, Memcached, for both read and write-heavy workloads.

Categories and Subject Descriptors

H.2.4 [**Systems**]: [Concurrency]

General Terms

Design, Performance

Keywords

key-value, cache, concurrency

1. INTRODUCTION

Extreme latency and throughput requirements have driven many large-scale application providers to store all active data in DRAM using in-memory key-value storage. In-memory, key-value (KV) stores and caches are a critical building block in many of today's large-scale web installations [3]. Memcached [5] is one of the most prominent examples with an impressive list of users which includes Facebook, Wikipedia, Twitter, and Youtube. As they gain widespread adoption, these stores will also become essential for real-time business analytics and decision support where high throughput and low latency are critical [21].

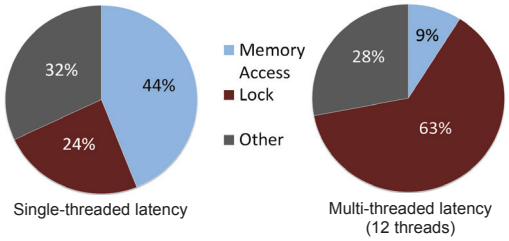


Figure 1: Latency breakdown of a Memcached GET.

Even though DRAM has high capital and operating costs, the effectiveness of in-memory key-value stores in utilizing available DRAM bandwidth has received little attention. The main focus has been on scaling-out through techniques such as sharding [5, 10], since per-node performance has been limited by comparably slow networks. However, recent advances in network technologies such as 10Gb/100Gb Ethernet, 40Gb Infiniband [7, 8], and RDMA now provide bandwidth matching DRAM and about one order of magnitude higher latency, shifting the bottleneck from the network to the memory subsystem [3, 6, 17]. Thus, it is critical to focus on micro-level aspects such as memory bandwidth utilization while employing macro-level scale-out methods for building high performance key-value stores.

Recently, Fan *et al.* proposed MemC3 [14], a re-design of Memcached internals to improve the memory bandwidth utilization. However, their design is suited for read-heavy workloads and it adversely impacts write performance by not being able to support concurrent writes. While this may be admissible for read-intensive workloads, such as those reported by Facebook [11], many emerging workloads exhibit much higher write-to-read ratio. Applications in many commercial spaces, such as real-time fraud detection, require a dataset to be continually updated with new information. This emerging class of use cases is focused on capturing, analyzing, and reacting to data in real-time. In order to understand this space further, we analyzed a commercial enterprise analytics workload that processes personal identifying information, such as name, address, and social-security number, in real-time. Every access to this database results in at least one and possibly many writes, since it functions by continually adding new observations to the database and checking for anomalies (*e.g.,* people with different names using the same social security number). Each time a new observation is made, writes are required. Although the exact read/write ratio is

SYSTOR '13, June 30 - July 02 2013, Haifa, Israel

dataset dependent, our trace analysis of a sample dataset showed a 1:1 read/write ratio.

Another trend is that application workloads are shifting towards smaller key and value sizes. Memcached pools sampled in [11], *e.g.,* user accounts, have 2B values and 16B or 21B keys. In our enterprise analytics workload, we observe that 78% of values are smaller than 8B and keys range from 9B to 57B. Thus, key-value access overhead is no longer masked by large values, which previously allowed relatively high memory bandwidth even with inefficient lookup strategies [21]. Small objects also increase the number of key-value pairs that can fit in the memory of a single server, rendering optimizations for the memory hierarchy, such as storing the hash table in CPU caches [19], ineffective.

Hence, we require a key-value store which takes into consideration these emerging workload characteristics and is able to efficiently utilize the available memory bandwidth. In this paper, we present a bottom-up approach to designing such a high performance in-memory key-value store, Mercury, suitable for both read-dominant and write-heavy workloads. We investigate the factors which impact the performance of a typical key-value store. Figure 1 demonstrates that DRAM accesses account for 44% of overall time spent servicing a GET request in a single-threaded instance of Memcached. Furthermore, as concurrency is increased, synchronization overheads dominate accounting for 63% of the overall latency. Based on these observations, we design Mercury to efficiently utilize the available resources, thereby, economizing [1] DRAM accesses per key-value operation and increasing concurrency by reducing synchronization overheads.

This paper makes the following contributions: (1) It demonstrates that a simple chaining-based hash table along with careful design of data structures (such as cache-conscious lock placement) can reduce DRAM accesses as compared to existing state-of-the-art techniques [17, 14]. Our evaluation indicates that Mercury incurs 31-46% fewer memory accesses than Memcached for read-only workloads. (2) It shows that a fine-grained, lightweight locking scheme enables Mercury to scale with increasing number of cores for both read-dominant and write-heavy workloads. Experiments with YCSB benchmark [13] reveal about 9.2x to 14x increase in throughput with our system as compared to Memcached for workloads with different read/write ratios.

2. DESIGN SPACE

In this section, we systematically evaluate the existing algorithms and data structures for building key-value stores and choose the approaches which meet our design principles.

2.1 Data Layout Methods

The techniques to leverage the CPU memory hierarchy [17, 19] are rendered ineffective for most large scale key-value workloads. These workloads not only have large working sets, both in terms of data volume and number of key-value pairs, but also demonstrate lack of locality and random access patterns [11]. Hence, attempts such as increasing cache hits by using a *constant*-size hash table that can fit into the CPU cache [19] may not be beneficial. For example, a 4GB key-value workload on a server with 12MB L3 cache will demonstrate a 99.7% probability of a cache miss for a random read request. Furthermore, it will require 2048 page-table entries when 2MB pages are used. In many modern architectures, the TLB size is not more than 128 entries, resulting in 0.94% probability of a TLB hit. Instead, decreasing the number of DRAM accesses per-operation allows us to reduce latency and improve throughput.

Adhering to our bottom-up approach, we consider two well accepted data layout methods: trees and hash tables, and analyze their memory bandwidth utilization. We define **memory bandwidth utilization** as the ratio of the amount of data requested by the application to the actual amount of data accessed at the memory controller. The ratio can be <1 as the memory cannot be accessed in units smaller than a CPU cache line. Given a value size V and a cache line size C, the maximum memory bandwidth utilization is $\frac{V}{C*\lceil V/C \rceil}$. For a 8B value, given a typical 64B cache line, the theoretical limit on the memory bandwidth utilization is 12.5%.

Tree-based layout. In tree-based key-value stores, the key and value pairs are stored in a tree. For analysis, we use the current state-of-the-art tree implementation from MassTree [17]. MassTree combines B^+ trees and tries, where every trie node is a B^+ tree. Each internal node in the B^+ tree stores constant sized key slices. The values (or the next layer pointers) are stored at the leaf nodes of each B^+ tree.

The potential number of DRAM accesses per operation is equal to the height h of the tree, as every lookup starts from the root. In the best-case (a balanced tree), $h = log_f(N)$, where f is the fanout and N is the number of key-value pairs stored. For N=100M and f=15, $h = \lceil log_f(N) \rceil = 7$. An additional memory access is required to retrieve the value. On average, MassTree can cache up to 4 levels of the tree in a trie[2], and retrieve 4 cache lines in one DRAM access. Thus, MassTree will make *at least 4 DRAM accesses* per operation: 3 accesses (each 256B) to retrieve the key and one access for the value. The total amount of memory accessed to retrieve an 8B value is $3 \cdot 256 + 64$ = 832 bytes, translating to only 0.96% the bandwidth utilization.

Hash-based layout. In a hash table implementation where the key-value pairs are stored separate from the table, the minimum number of DRAM accesses per lookup is 2: one to access the hash table and one to access the key-value pair. DRAM accesses can be saved by storing key-value pairs within the hash table. However, this implies that the hash table width needs to be as large as the largest key-value pair size. This can result in severe memory wastage, especially when most of the value sizes are small while only a few values are large. The scalable approach, therefore, is to store key-value pairs separate from the hash-table.

Memcached [5] and many key-value stores based on it [10, 19] use chained hashing. A key-value pair is retrieved by first hashing the key to locate the correct hash table bucket, and then retrieving the data by traversing the key-value pair list. Thus, the number of DRAM accesses is $D = 2 + X$, where X is additional key-value pairs traversed due to collision.

Another approach to hash table implementation is open addressing which stores a single key-value pair per bucket. Collisions are resolved through approaches such as linear probing and quadratic probing [15]. MemC3 [14] uses set associative cuckoo hashing [16], a form of open-addressing. It can achieve 3.03 or 4.03 DRAM accesses per GET operation: 2.03 to 3.03 DRAM accesses to retrieve the key and one DRAM access for the value [14]. However, it optimizes reads at the cost of writes. Adding a new key-value pair in MemC3 can displace multiple existing key-value pairs in the hash-table, causing multiple DRAM accesses. Moreover, cuckoo hashing locks the entire hash table while expanding, causing degraded performance during expansion [2] which can be critical for write-heavy workloads.

[1] We use the term economizing to imply *as low as possible* without the implication of a mathematical minimization function.

[2] Consider that 1MB of L3 cache, caches the interior nodes. As the interior node size is 277B in MassTree, 3785 nodes can be stored in the L3 cache, which corresponds to 4 levels at f=15 and N=100 million.

Design Choice: Based on the above analysis, MERCURY uses chained hashing with an aggressive hash table expansion mechanism to ensure that chain lengths remains short (near 1), thus keeping the number of DRAM accesses close to 2 for both reads and writes. For small key-value pairs this often translates to two cache line accesses. Thus, for an 8B value, the *practical limit* on the memory bandwidth utilization, given a typical 64B cache line is $\frac{8}{2 \cdot 64} = 6.25\%$ (recall that theoretical limit is 12.5% in this case), which is about 6.5 times as can be achieved by a tree-based method.

2.2 Concurrency

High concurrency enables better memory bandwidth utilization on multi-core hardware and requires reducing synchronization overheads. Existing concurrent hashmap approaches [20, 1] use a fixed collection of locks, where each lock protects a set of buckets, enabling concurrent writes. However, for large datasets, this coarse-grained locking approach can result in contention among threads performing concurrent writes on different buckets protected by the same lock. Furthermore, these write operations require access to a shared array of locks, incurring cache invalidations on multi-core hardware [12]. This significantly reduces multi-threaded throughput for write-heavy workloads. Hash table wide operations such as hash table expansion require exclusive access to all the locks, further degrading performance.

Another set of approaches supports highly concurrent accesses for read-intensive workloads. MemC3 uses cuckoo-hashing combined with optimistic locking to scale read throughput. However, it only supports a single writer at a time impacting write concurrency. Java's ConcurrentHashMap allows retrieval operations to proceed without acquiring a lock and synchronizes on detecting stale data [4]. For workloads with high write-to-read ratio, this optimistic approach can increase access overheads. Furthermore, removal of an item requires cloning all or part of the bucket storing the item, thus reducing concurrency. Lock-free expandable hash tables [22, 23] have been proposed but are either not space-efficient or their performance depends on cache locality, thus impacting their general applicability.

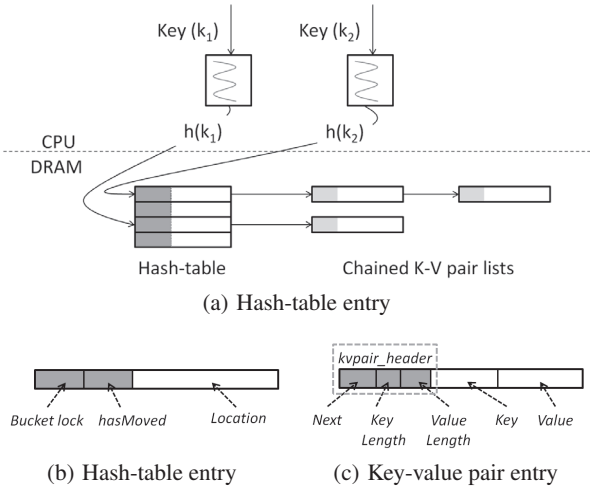
Many key-value systems reduce synchronization overheads by using static partitioning of data between multiple isolated instances running on the same machine. However, this can cause load imbalance in the presence of skewed access patterns exhibited by real-world workloads [14, 17]. The goal of this paper is to reduce per-instance synchronization overheads, but Mercury's performance may be further improved with partitioning techniques in the future.

Design Choice: Mercury uses fine-grained locking - lock per bucket - along with aggressive hash table expansion to reduce chaining. A cache-conscious placement of these locks enables us to reduce cache invalidations, achieving high concurrency and scaling with the number of cores.

3. MERCURY DESIGN

At Mercury's core is a chained hash table [18], as shown in figure 2(a). Each entry in the hash-table contains a pointer to the list of key-value pairs that hash to the same bucket. The key-value structure is illustrated in figure 2(b). We utilize two techniques to economize DRAM accesses per-operation: cache-conscious placement of locks and aggressive hash table expansion for keeping chain lengths small. We also develop a simple mechanism for reducing contention for the memory allocator/de-allocator.

Locks. To achieve high concurrency, Mercury implements fine-grained locking, where each hash table bucket with its key-value pair list is protected by its own lock. Thus, if there are N buckets



(a) Hash-table entry

(b) Hash-table entry (c) Key-value pair entry

Figure 2: MERCURY's data structures.

in the hash table, there are N locks. Once a thread acquires a lock, it has access to all the key-value pairs within that bucket.

Accessing a lock may incur additional cache misses [12] because locks are stored in DRAM and are shared by multiple threads. We eliminates these penalties by using cache-conscious placement of locks: A lock protecting a given hash table bucket (*bucket-lock*), is co-located with the hash table entry (Figure 2(b)). This ensures that accessing a hash table entry results in no more than one cache miss, because fetching the entry also brings in the lock.

Hash table expansion. Mercury provides aggressive hash table expansion that aims to keep the average number of key-value pairs per bucket near 1. This helps achieve close to minimum number of DRAM accesses per-operation. To reduce expansion's negative impact on performance, we re-hash existing keys one bucket at a time. Expansion is carried out in three phases. During the **initialization** phase, the hash table state is changed from non-expanding to *expanding*, and a new hash table is initialized. The actual **re-hashing** phase re-hashes all the keys from the existing hash table to the new one. A dedicated thread, called *expand-thread*, acquires a single bucket lock, re-hashes all the keys in that bucket into the new hash table, and then releases the lock. While the expand-thread is re-hashing an existing key, other threads can freely access other buckets and perform operations on key-value pairs. Because of aggressive expansion, chain lengths are small. Thus, the time for re-hashing keys from each bucket is low. This allows the bucket to be available quickly for servicing requests. Finally, the **termination** phase resets the hash-table state to *non-expanding* and the memory for the old hash table is freed.

Unlike traditional expansion techniques which require exclusive access during the entire re-hashing process, MERCURY requires global synchronization only during initialization and termination phases because of its fine-grained locking mechanism. This exclusive access is needed to prevent a possible race condition, when the incorrect hash table is accessed by threads performing key-value operations (*worker-threads*) during the change in the hash table state. We address this problem by providing each worker-thread with its own *thread-lock* which it shares with the expand-thread. This lock is acquired by each worker-thread before performing any operation and by the expand-thread during its state change operation. The synchronization overhead is extremely small since these locks are only contended during hash table state change which requires less than 1μsec according to our measurements. Further-

more, since every worker-thread has its own thread-lock, there is no contention during non-expansion phase.

During re-hashing phase, worker-threads need to decide *which* hash table to access during a lookup - the existing or the hash table being created. We address this problem by keeping track of the number of buckets re-hashed by the expand-thread. Suppose, the existing hash table has N buckets ($2 * N$ buckets in the new hash table). The expand-thread has re-hashed r buckets ($-1 \leq r < N$) and is currently re-hashing the keys in the $(r+1)^{th}$ bucket. A worker-thread performing a key-value operation determines the appropriate bucket for the given key in the existing hash table. Suppose the key hashes to bucket l in the existing hash-table: $l = hash(key)\%N$, where $0 \leq l < N$. There are two invariants. (i) ($l < r$): The expand-thread has already re-hashed the key, so the lookup will use the new hash table. The index in the new hash table will be $l\prime = hash(key)\%(2N)$. (ii) ($l \geq r$): The expand-thread is yet to re-hash the given key, so the lookup will use the existing hash table.

A race condition can happen if the key hashes to a bucket which is currently being re-hashed by expand-thread. In this scenario, the worker-thread will wait on the bucket lock while the expand-thread does its re-hashing. Once the bucket lock is released by expand-thread, the worker-thread will acquire it even though the key-value pairs have been migrated to the new hash table. We avoid this situation by introducing a *hasmoved* bit in hash table entries (Figure 2(b)). Whenever the expand-thread moves the key-value pairs in a bucket, it marks the *hasmoved* bit. When the worker thread acquires the bucket lock, it checks for the *hasmoved* bit. If it is marked, the worker thread performs the lookup in the new hash table. The co-location of the *hasmoved* bit with a hash table entry enables zero cache miss penalty.

Memory Allocator: The memory management unit is responsible for allocating the memory to new key-value pairs and de-allocating the memory from the deleted key-value pairs. Mercury employs pool-based memory allocation where every thread has its own local pool of memory. Threads request memory from the manager in large chunks (such as 2MB) and add it to their pool. Subsequently, memory allocation/de-allocation requests are satisfied from these local pools without contention for any locks. We plan to explore more sophisticated memory allocator designs in our future work.

4. EVALUATION

In this section, we evaluate the impact of our design choices for Mercury and compare its performance to Memcached (v1.4.13) which is a widely deployed open-source key-value cache. This provides fair comparison, because both systems use chained hashing, while employing different locking and hash table expansion techniques.

Setup. Our experiments run on a 12-core single socket 2.67GHz (Intel Xeon X5650) server with 24GB DRAM running Linux 3.2.0. Each core has 64KB L1 cache and 256KB L2 cache. All the cores share 12MB L3 cache. The cache line size is 64B. We enable 2MB hugepage support to reduce TLB misses. To exclude the network overhead, we evaluate both systems without the networking component.

4.1 Microbenchmarks

We use microbenchmarks to evaluate the memory bandwidth utilization and throughput achieved by Mercury and compare it to Memcached. The workload consists of 100% GET requests that randomly access 32 million pre-loaded key-value pairs, consisting of 8B keys and values each. We observe similar trends for workloads with larger values, but omit them due to space constraints.

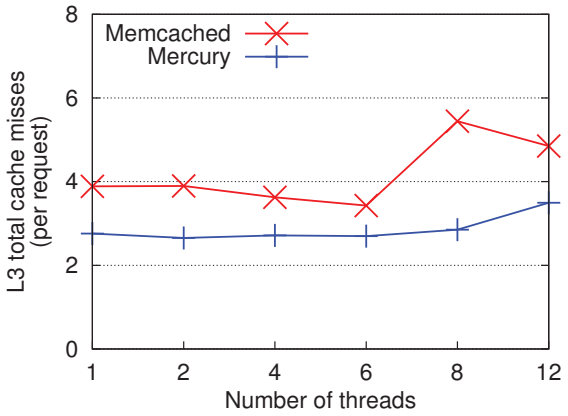


Figure 3: L3 misses per GET request.

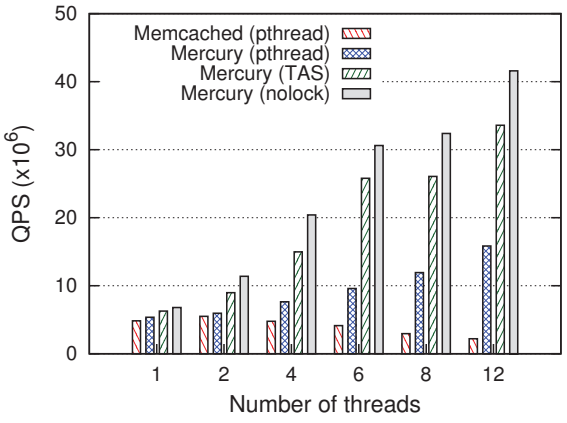
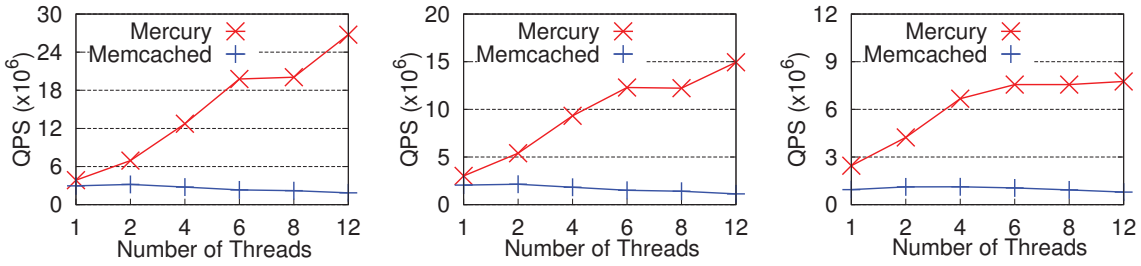


Figure 4: GET throughput using different locking schemes in MERCURY and Memcached.

Memory bandwidth utilization. We evaluate memory bandwidth utilization of Mercury and Memcached by comparing cache penalties. All the hardware measurements are performed using Performance API (PAPI) [9].

Figure 3 shows the average number L3 misses (represents DRAM accesses) incurred for each GET operation as we vary the number of threads. We observe that Mercury makes about 1.2 fewer DRAM accesses on average for each GET request as compared to Memcached for single-threaded instances. In fact, when we increase concurrency to 8 threads, Memcached's DRAM accesses per request increase to 5.44 while number of DRAM accesses in Mercury remains almost the same at 2.85. This can be attributed to cache-conscious placement of locks and aggressive expansion (during the initial key-value insertion phase) which keeps the chain lengths small. Since our chain lengths are not always 1, our average number of DRAM accesses per GET operation is greater than 2, resulting in memory bandwidth utilization which is $2/2.85 = 70\%$ of the practical limit. On the other hand, memory bandwidth utilization of Memcached is $2/5.44 = 36.7\%$ at 8-threads.

Furthermore, we observe that L1 and L2 misses incurred by Mercury are always less than Memcached by up to 34% and 53% respectively (not shown due to space constraints). This is primarily



(a) 100% GET (b) 75% GET (c) 50% GET

Figure 5: Throughput of YCSB workloads

due to per-bucket locks in Mercury which reduces the number of cache invalidations in SMPs as compared to coarse-grained locks in Memcached.

Throughput. Figure 4 shows the throughput in terms of queries per second (QPS) achieved by different configurations of Mercury and Memcached as the number of threads increases. Mercury (TAS) represents our Mercury implementation. It uses test-and-set (TAS) based locks. For categorizing the upper bound on Mercury's performance, we also present the results of Mercury where all locks are removed, Mercury (nolock). While this configuration provides the best performance, it does not ensure correctness. Memcached (pthread) uses pthread-based locks, which is Memcached's default configuration. For fair comparison, we also provide the results for Mercury using pthread locks, Mercury (pthread). Memcached (TAS) is omitted because we do not observe benefits of TAS locks in Memcached.

For a single-threaded instance, Mercury outperforms Memcached by 16%, which also shows improvement in GET latency. As we increase the number of threads, the gap in performance between the two systems starts to widen. In fact, Memcached's throughput starts to degrade as concurrency is increased, whereas Mercury's performance scales with the number of threads. With 12 threads, there is more than 15x gap in the performance of the two systems. This illustrates the benefits of fine-grained locking over the fixed number of locks used by Memcached. Furthermore, these techniques allow us to achieve throughput close to the no-lock implementation. We plan to explore mechanisms to further reduce this gap as part of our future work.

We also evaluate the impact of aggressive hash table expansion on throughput. For 8 threads, the throughput of MERCURY *during expansion* is only about 9% less than the throughput observed without expansion (not shown due to space constraints). This shows that expansion incurs a small overhead while providing significant benefits by economizing DRAM accesses.

4.2 YCSB Benchmark

This section evaluates mixed GET/PUT performance using workloads generated with the Yahoo! Cloud Serving Benchmark (YCSB) [13]. We vary the percentage of PUT requests to simulate emerging workloads with varying write-to-read ratio. The key and the value sizes are set to 24B and 10B respectively. In each experiment, the key-value store is initialized with 24 million key-value pairs and the benchmark is executed for 100 million requests generated using Zipfian popularity distribution. The number of columns was set to one as Memcached and Mercury do not support columns.

Figure 5 shows that Mercury outperforms Memcached for all workloads with different write-to-read ratio with throughput higher by an order of magnitude ranging from 9.2x to 14x for 12 threads. Even for workloads with 50% writes in them (Figure 5(c)), Mercury's performance scales up to number of physical cores (6) in the system while Memcached fails to scale with increased concurrency. This clearly demonstrates that careful design of data structures to exploit available memory bandwidth can have a huge impact on the performance and scalability of a key-value store.

5. FUTURE WORK

In this work, we outlined the core components of a key-value store designed to make efficient use of underlying hardware. In section 3, we described a simple mechanism to reduce contention while allocating/de-allocating memory for key-value pairs. We plan to extend this mechanism to build a scalable memory allocator for Mercury. Another critical aspect of in-memory key-value stores (beyond caches) is the need for data persistence. Thus, going forward, we will look into different software techniques and devices to enable a high performance and scalable backend for Mercury. We will investigate the consistency-performance tradeoffs in such an environment and study their impact on different enterprise-scale workloads.

6. CONCLUSION

In this paper, we systematically explored the design choices for building a highly efficient key-value store for both traditional read-dominant and emerging write-intensive workloads. We observed that a simple chained hashing with fine-grained, cache-friendly locks and aggressive, but efficient re-hashing scheme is capable of reducing DRAM accesses and improving throughput by about 14x as compared to Memcached.

7. REFERENCES

[1] Class concurrenthashmap. `http://gee.cs.oswego.edu/dl/jsr166/dist/docs/java/util/concurrent/ConcurrentHashMap.html`.

[2] Cuckoo hashing expansion. `http://users.cis.fiu.edu/~weiss/dsaajava3/code`.

[3] High performance at massive scale - Lessons learned at Facebook. `http://tinyurl.com/ygvkpet`.

[4] Java theory and practice: Building a better hashmap. `http://www.ibm.com/developerworks/library/j-jtp08223/`.

[5] Memcached. `http://memcached.org`.

[6] Memcached multiget hole. `http://tinyurl.com/yz5wyvc`.

[7] Network bandwidth growth. http://tinyurl.com/9hx23su.

[8] Network bandwidth growth. http://tinyurl.com/9n4g34u.

[9] "papi". http://icl.cs.utk.edu/papi/.

[10] Redis. redis.io.

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, et al. Workload analysis of a large-scale key-value store. SIGMETRICS '12, pages 53–64, 2012.

[12] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. OSDI'10, pages 1–8, 2010.

[13] B. Cooper, A. Silberstein, E. Tam, et al. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[14] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. 10th USENIX NSDI*, Lombard, IL, Apr. 2013.

[15] G. D. Knott. Expandable open addressing hash table storage and retrieval. SIGFIDET '71, pages 187–206, 1971.

[16] H. Lim, B. Fan, D. G. Andersen, et al. Silt: a memory-efficient, high-performance key-value store. SOSP '11, 2011.

[17] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. EuroSys '12, pages 183–196, 2012.

[18] W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Comput. Surv.*, pages 5–19, 1975.

[19] Z. Metreveli, N. Zeldovich, et al. Cphash: a cache-partitioned hash table. PPoPP '12, pages 319–320, 2012.

[20] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proc. 10th USENIX NSDI*, Lombard, IL, Apr. 2013.

[21] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[22] O. Shalev and N. Shavit. Split-ordered lists: lock-free extensible hash tables. PODC '03, pages 102–111, 2003.

[23] D. Zhang and P.-A. Larson. Lhlf: lock-free linear hashing (poster paper). PPoPP '12, 2012.