# An In-Memory Object Caching Framework with Adaptive Load Balancing

Yue Cheng⋆, Aayush Gupta†, Ali R. Butt⋆

⋆Virginia Tech, †IBM Research – Almaden

{yuec,butta}@cs.vt.edu, {guptaaa}@us.ibm.com

## Abstract

The extreme latency and throughput requirements of modern web applications are driving the use of distributed in-memory object caches such as Memcached. While extant caching systems scale-out seamlessly, their use in the cloud — with its unique cost and multi-tenancy dynamics — presents unique opportunities and design challenges.

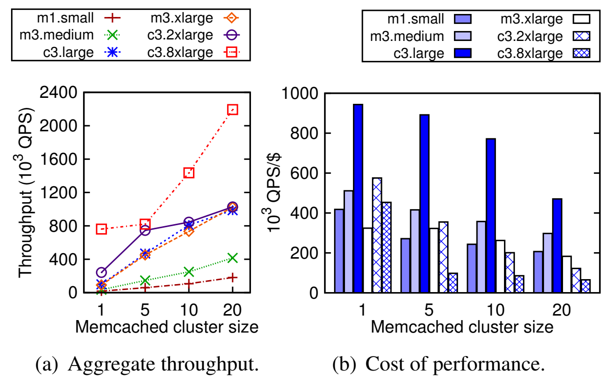
In this paper, we propose MBal, a high-performance in-memory object caching framework with adaptive Multiphase load Balancing, which supports not only horizontal (scale-out) but vertical (scale-up) scalability as well. MBal is able to make efficient use of available resources in the cloud through its fine-grained, partitioned, lockless design. This design also lends itself naturally to provide adaptive load balancing both within a server and across the cache cluster through an event-driven, multi-phased load balancer. While individual load balancing approaches are being leveraged in in-memory caches, MBal goes beyond the extant systems and offers a holistic solution wherein the load balancing model tracks hotspots and applies different strategies based on imbalance severity – key replication, server-local or cross-server coordinated data migration. Performance evaluation on an 8-core commodity server shows that compared to a state-of-the-art approach, MBal scales with number of cores and executes 2.3× and 12× more queries/second for GET and SET operations, respectively.

## 1. Introduction

Distributed key-value stores/caches have become the *sine qua non* for supporting today's large-scale web services. Memcached [6], a prominent in-memory key-value cache, has an impressive list of users including Facebook, Wikipedia, Twitter and YouTube. It can scale to hundreds of nodes, and in most cases, services more than 90% of database-backed queries for high performance I/Os [12, 40, 46].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*EuroSys'15*, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/2741948.2741967




(a) Aggregate throughput. (b) Cost of performance.

**Figure 1:** Aggregated peak throughput and KQPS/$ observed for different Amazon EC2 cluster configurations. Here, the network or CPU of Memcached cluster is saturated to achieve the highest throughput under each configuration for a 95% GET workload.

With the growth of cloud platforms and services, in-memory caching solutions have also found their way into both public and private clouds. In fact, cloud service providers such as Amazon, IBM Cloud and Google App Engine, already support in-memory caching as a service. Amazon's ElastiCache [2] is an automated Memcached deployment and management service widely used by cloud-scale web applications, e.g., Airbnb, and TicketLeap.

While the cloud model makes in-memory caching solutions easy to deploy and scale, the pay-as-you-go approach leads to an important consideration for the cloud tenant: *How do I (the tenant) get the most bang-for-the-buck with in-memory caching deployment in a shared multi-tenant environment?* To understand the different aspects of this issue, we need to consider two aspects:

***Impact of Resource Provisioning*** A key promise of the cloud model is to offer the users choice in terms of resources, services, performance, cost, tenancy, etc. In order to better understand the impact of different options in a true multi-tenant environment, we conducted an experimental study

| Instance type | VCPUs (cores) | Memory (GB) | Network (Gbps) | Cost ($/hr) |
|---|---|---|---|---|
| `m1.small` | 1 | 1.7 | 0.1 | 0.044 |
| `m3.medium` | 1 | 3.75 | 0.5 | 0.07 |
| `c3.large` | 2 | 3.75 | 0.6 | 0.105 |
| `m3.xlarge` | 4 | 15 | 0.7 | 0.28 |
| `c3.2xlarge` | 8 | 15 | 1 | 0.42 |
| `c3.8xlarge` | 32 | 60 | 10 | 1.68 |

**Table 1:** Amazon EC2 instance details based on US West – Oregon, Oct. 10, 2014 [1] (network b/w measured using Netperf).

on Amazon EC2 public cloud. Figure 1 demonstrates the impact of scaling cluster size on Memcached performance with respect to different resource types (EC2 instances).

Figure 1(a) shows the impact on raw performance (reflected by kilo Queries Per Second (KQPS)), and Figure 1(b) captures the effective cost of performance by normalizing the performance with the cost of the corresponding EC instances (KQPS/$). The figures show that there is a low return on investment for powerful instances such as `c3.8xlarge` compared to relatively cheap instances such as `c3.large`. The extreme points behave as expected with the smaller-capacity instances (`m1.small`, `m3.medium`) achieving much lower throughput compared to larger-capacity instance. However, we observe that the performance of the three semi-powerful instance types (`c3.large`, `m3.xlarge`, and `c3.2xlarge`) converges to about 1.1 MQPS (million QPS) as the cluster size (for each instance type) scales to 20 nodes. We believe that this behavior can be attributed to constrained network bandwidth because of the following reasons. (1) Even though these instances have different CPU capacities (as shown in Table 1), they all have similar network connectivity with an upper bound of 1 Gbps. (2) The underlying cluster or rack switches might become the bottleneck due to incast congestion [42] under Memcached's many-to-many network connection model. (3) Increasing the number of clients does not change performance. (4) The server CPUs, as observed, have a lot of free cycles in the semi-powerful instance types. For example, while CPU utilization in the `m1.small` setup was close to 100% (bounding the performance), the `c3.2xlarge` cluster setup had about 40% free cycles available. (5) Finally, improving network bandwidth to 10 Gbps (`c3.8xlarge`), doubles the throughput. This clearly shows that the performance of these semi-powerful instances is constrained by the available network bandwidth. However, even the performance of the most powerful `c3.8xlarge` instance that we tested, does not scale well with the increase in resource capacity (and monetary cost). This may be due to the multi-tenant nature of the public cloud where tenants or even virtual machines of the same tenant co-located on a host may indirectly interfere with each other's performance.

From our experiments, we infer the following. (i) While cost in the cloud scales linearly with the cluster size, the performance does not, causing the overall performance-to-cost efficiency to come down. (ii) Unlike private data centers where we typically observe large scale-out with powerful machines [12] for in-memory caching tier, cloud tenants are faced with the "problem of plentiful choices" in the form of different configurations and their impact on workloads, when deploying their in-memory caching tier. This in turn increases the number of variables tenants have to consider while making deployment decisions, and is burdensome to the extent that tenants typically choose the "easy-but-inefficient" default parameters. (iii) Our study shows that tenants may be better served by deploying moderate scale clusters with just enough CPU and memory capacity to meet their requirements to get best cost-performance ratio[1]. *This stresses the need for the caching software to make efficient use of available resources.*

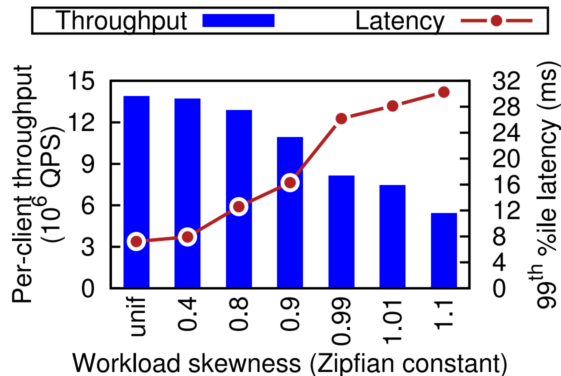



**Figure 2:** Impact of skewness on the performance of a Memcached cluster with 20 instances. A larger Zipfian constant implies a more skewed workload. The workload is 95% GET, generated using Yahoo! Cloud Serving Benchmark [15] (YCSB) with 12 clients.

***Impact of Load Imbalance*** Facebook's memcache workload analysis [12] reports high access skew and time varying request patterns, implying existence of imbalance in datacenter-scale production deployments. This load imbalance is significantly amplified — by orders of magnitude — on cloud-based cache deployments, due to a number of reasons including key popularity skewness [40], multi-tenant resource sharing, and limited network bandwidth.

To quantify the impact of load imbalance, we measured the throughput and latency of a typical read-intensive workload (95% GET) with varying load skewness (represented by Zipfian constant). Figure 2 shows that the performance declines as the workload skewness increases (`unif` represents uniform load distribution). We observe that hotspots due to skewness can cause as much as $3\times$ increase in the $99^{th}$ percentile tail latency and more than 60% degradation in average per-client throughput. Similar results have been observed by Hong et al. [23]. Thus, efficient load balanc-

[1] Finding the best combination of instance types for cloud workloads is beyond the scope of this paper and is part of our future work.

ing mechanism in the caching tier is necessary for providing high performance and efficient resource (CPU, memory) utilization.

***Contributions*** Based on the two requirements of efficient use of resources and handling load imbalance in cloud-based in-memory cache deployments, we develop MBal, an in-memory object caching framework that leverages fine-grained data partitioning and adaptive Multi-phase load Balancing. MBal performs fast, lockless inserts (SET) and lookups (GET) by partitioning user objects and compute/memory resources into non-overlapping subsets called *cachelets*. It quickly detects presence of hotspots in the workloads and uses an adaptive, multi-phase load balancing approach to mitigate any load imbalance. The cachelet-based design of MBal provides a natural abstraction for object migration both within a server and across servers in a cohesive manner.

Specifically, we make the following contributions:

1. We evaluate the impact of co-located multi-tenant cloud environment on cost of performance by conducting experiments on Amazon EC2 based cloud instance. Our observations stress *the need to carefully evaluate the various resource assignment choices available to the tenants and develop simple rules-of-thumb that users can leverage for provisioning their memory caching tier*.
2. Based on our behavior analysis of Memcached in the cloud, we design and implement a fine-grained, partitioned, lockless in-memory caching sub-system MBal, which *offers improved performance and natural support for load balancing*.
3. We implement an adaptive load balancer within MBal that (1) *determines* the extent of load imbalance, and (2) uniquely *applies* local, decentralized as well as globally coordinated load balancing techniques, to (3) *cost-effectively* mitigate hotspots. While some of the load balancing techniques have been used before, MBal *synthesizes the various techniques into a novel holistic system and automates the application of appropriate load balancing* as needed.
4. We deploy our MBal prototype in a public cloud environment (Amazon EC2) and validate the design using comprehensive experimental evaluation on a 20-node cache cluster. Our results show that MBal agilely adapts based on workload behaviors and achieves 35% and 20% improvement in tail latency and throughput.

***Roadmap*** The rest of the paper is organized as follows. We describe the overall design of MBal in §2. §3 presents our load balancing framework. We evaluate the benefits of our design in §4, and discuss related work in §5. Finally, we present our conclusions in §6.

## 2. MBal Architecture

We design MBal in light of the requirements of cloud-based deployments – efficient use of available resources and need for load balancing. Conventional object caches/stores, such as Memcached [6], use a monolithic storage architecture where key space sharding is performed at coarse server granularity while resources within an object server are shared across threads. This design has good scale-out characteristics, as demonstrated by Memcached deployments with hundreds of servers [3, 40], but is not necessarily resource efficient. For example, a known and crucial problem in Memcached is that it suffers from global lock contention, resulting in poor performance on a single server.

To address this issue, MBal performs fine-grained thread-level resource partitioning, allowing each thread within a cache server to run as a fully-functional caching unit while leveraging the benefits of running within a single address space. While the concept of thread-level resource partitioning has been explored [26, 30, 36], the approach provides significant benefits for a fast DRAM-based cache. This allows MBal to not only *scale-out* to a large number of cache nodes similar to its contemporary counterparts but also *scale-up* its performance by fully exploiting the parallelism offered by multi-core architectures. Furthermore, thread-level resource partitioning provides the ability to perform low overhead load balancing.

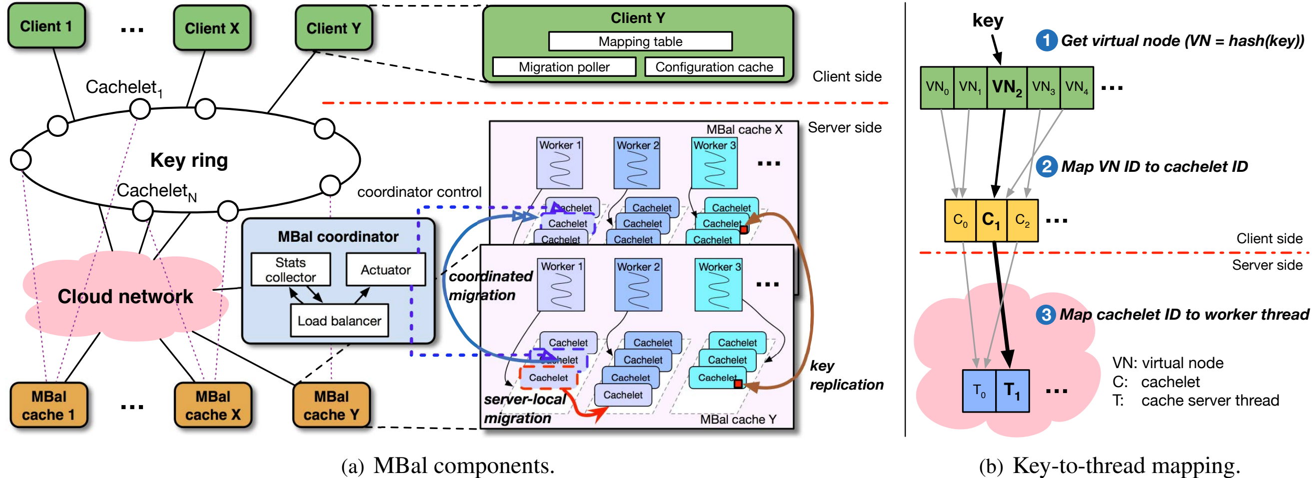
### 2.1 Cachelet Design

Typical in-memory object caches use consistent hashing [31] to map keys (object handles) to cache servers. The sharding process involves mapping subsets of key space to *virtual nodes* (VN) and mapping VNs to cache servers. This allows distributing non-consecutive key hashes to a server. However, the cache servers are typically unaware of the VNs.

We introduce a new abstraction, *cachelets*, to enable server worker threads to manage key space at finer granularity than a monolithic data structure. A cachelet is a configurable resource container that encapsulates multiple VNs and is managed as a separate entity by a single worker thread. As depicted in Figure 3(a), each worker thread in a cache server owns one or more cachelets. While the design permits one-to-one mapping between VNs and cachelets, typically there can be an order(s) of magnitude more VNs than cachelets. The choice is based on the client administrator's desired number of subsets of key space and the speed at which the load balancing algorithm should converge. To this end, cachelets help in decoupling metadata management at the servers/clients and provide resource isolation.

### 2.2 Lockless Operations

Each cachelet is bound to a single server worker thread that allocates memory, manages accesses, and maintains metadata structures and statistics for the cachelet. This partitioning ensures that in MBal, there is no lock contention or syn-




(a) MBal components. (b) Key-to-thread mapping.

**Figure 3:** MBal architecture. Clients use the three-step key-to-thread mapping to route requests to MBal servers. Key replication involves copying hot keys across servers, whereas both server-local and coordinated migration perform load balancing at cachelet granularity.

chronization overheads across worker threads during inserts or lookups. Furthermore, this allows MBal to reduce false sharing by cross-thread resource isolation. The design is also amenable to and provides a mechanisms to quickly serialize and migrate data for load balancing (server-local and coordinated migration in Figure 3(a)). In the future, we aim to add functionality to cachelets such as service differentiation and server-side code execution [11], which will enable MBal to support richer services beyond object caching.

### 2.3 Key-to-Thread Mapping

A naive approach for routing a request for an object to an appropriate worker thread on a server is to use a server-side dispatcher thread: A dedicated thread on each MBal server receives client requests and dispatches the request to an appropriate worker thread based on cachelet ID in the request. We first implemented our design using this approach and quickly found the dispatcher thread to be a bottleneck. Increasing the number of dispatcher threads reduces the number of cores available on a server to service requests but does not improve performance, and thus is impractical.

To avoid this, MBal provides client-side routing capability within MBal's client library, similar to approaches used in mcrouter [40]. We associate a TCP/UDP port with each cache server worker thread so that clients can directly interact with workers without any centralized component. As shown in Figure 3(b), this approach performs "on-the-way-routing" via a two-level mapping table lookup on the client.

The mapping scheme enables convenient mapping changes when servers perform cachelet migration. In our implementation, we overload the field originally reserved for virtual bucket [7] in Memcached protocol header to hold cachelet ID. Thus, no client application changes are needed and web applications can easily work with MBal cache by simply linking against our Memcached protocol compliant client library. Finally, assigning a separate network port to each worker thread on a server is not a concern. This is because while the number of worker threads depends on user configuration, usually it is expected to be the same as the number of cores on the cache server machine. Note that having too many worker threads on a server can lead to significant network interrupt overhead due to request overwhelming [40] as well as cross-socket cache coherence traffic [38]. MBal also employs cache-conscious bucket lock placement suggested by [21] to reduce additional last-level cache (LLC) misses. Each bucket lock protecting one hash table bucket is co-located with that hash table entry that is cache-line-aligned. This guarantees that a hash table access results in at most one cache miss.

### 2.4 Memory Management

MBal employs hierarchical memory management using a global memory pool and thread-local memory pool managed using a slab allocator [14, 18]. Each worker thread requests memory from the global free pool in large chunks (configurable parameter) and adds the memory to its local free memory pool. New objects are allocated from thread-local slabs and object deletes return memory to thread's own pool for reuse. Furthermore, workers return memory to global heap to be reused by other threads if global free pool shrinks below a low threshold (`GLOB_MEM_LOW_THRESH`) and thread's local free pool exceeds a high threshold (`THR_MEM_HIGH_THRESH`). Such allocation reduces contention on the global heap during critical insert/delete paths and localizes memory accesses. Besides, the approach also provides high throughput when objects are evicted (deleted) from the cache to create space for new ones. MBal uses LRU replacement algorithm similar to Memcached but aims to provide much better performance by reducing lock contention. Additionally, MBal adds

support for Non-Uniform Memory Access (NUMA) aware memory allocator in the thread-local memory manager.

### 2.5 Discussion: Multi-threading vs. Multi-instance

An alternative to the multi-threaded approach in MBal is to use multiple single-threaded cache server instances that can potentially achieve good resource utilization. Such single-threaded instances can even be binded to CPU cores to reduce the overheads of process context switches. While intuitive, such a multi-instance approach is not the best design option in our opinion because of the following reasons. (1) An off-the-shelf multi-instance implementation (e.g., run multiple single-threaded cache instances) would require either static memory allocation or a dynamic per-request memory allocation (e.g., using `malloc()`). While the former approach can lead to resource under-utilization, the latter results in performance degradation due to overheads of dynamic memory allocation. (2) While possible, hierarchical memory management is costly to implement across address spaces. A multi-threaded approach allows memory to be easily rebalanced across worker threads sharing the single address space, whereas a multi-instance approach requires using shared memory (e.g., global heap in shared memory). Such sharing of memory across processes is undesirable, especially for writes and non-cache-line-aligned reads, as each such operation may suffer a TLB flush per process instead of just one in the multi-threaded case. (3) Multi-instance approach entails costly communication through either shared memory or inter-process communication, which is more expensive compared to MBal's inter-thread communication for server-local cachelet migration. (4) Recent works, e.g., from Twitter [4], have shown that multi-instance deployment makes cluster management more difficult. For example, in a cluster where each machine is provisioned with four instances, the multi-instance deployment quadruples management cost such as global monitoring. (5) Finally, emerging cloud operating systems [33, 39] are optimized for multi-threaded applications with some, such as OS$^v$ [33], supporting a single address space per virtual machine. Consequently, we have designed MBal while considering all the benefits of multi-threaded approach, as well as future portability of the system.

## 3. Multi-Phase Load Balancing

MBal offers an adaptive load balancing approach that comprises different phases, each with its unique efficacy and cost. In the following, we describe these load balancing mechanisms and the design choices therein.

### 3.1 Cluster-Wide Multi-Phase Cost/Benefit Analyzer

A distributed in-memory object cache can be load balanced using a number of techniques, such as key replication and data migration. However, each technique has a unique cost associated with it, thus requiring careful analysis to choose what technique to employ and when. The more expensive approaches typically yield better load balancing. However, such a heavy-weight approach, e.g., cross-server data migration, may not always be justified if the system is imbalanced due to a small set of hot keys. Consequently, the load balancer should consider current workload in deciding what approach to use so as to ensure high efficiency.

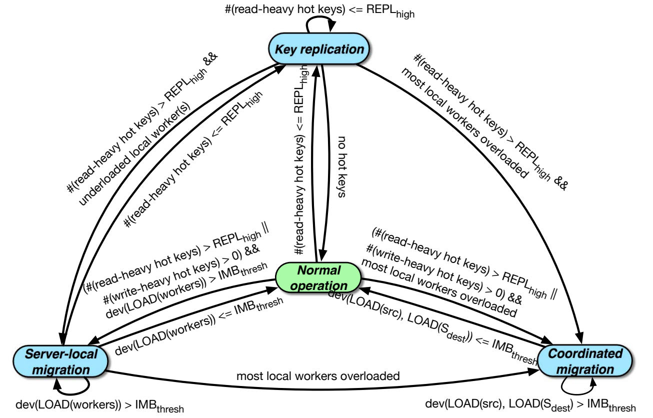



**Figure 4:** The state transitions of the MBal load balancer at each server.

MBal employs an event-driven multi-phase load balancer, where each phase corresponds to a series of decision making processes triggered by event(s) based on key access patterns. A phase may also be triggered and executed simultaneously with another lower-priority phase. For example, if the configurable high replication watermark ($REPL_{high}$) is exceeded, a worker may lower its priority on key replication by reducing the key sampling rate and thus the replication overhead. The worker may then simultaneously enter another phase, e.g., server-local cachelet migration. The goal is to generate load balancing plans that are fast and economical, yet effective for a given workload. MBal implements three phases: (1) key replication; (2) server-local cachelet migration; and (3) coordinated cachelet migration. While the first two phases are locally implemented at each MBal server providing quick solutions for ephemeral hotspots, the third phase involves a centralized coordinator to address any longer-term persistent load imbalances. Table 2 lists the salient properties of each of the phases, and describes the associated costs and benefits that we consider in MBal. The techniques used in each of the MBal phases are beginning to be employed in different existing systems, but individually and in an ad hoc fashion. The novelty of MBal lies in the coherent synthesis of the various load balancing techniques into a holistic design that offers an automated end-to-end system.

Figure 4 shows the state machine for seamlessly transitioning between the different phases of our load balancer. Each MBal server monitors the state of local workers by keeping track of both: (1) object access metrics (reads and

| Phase | Key/Object replication (§3.2) | Server-local cachelet migration (§3.3) | Coordinated cachelet migration (§3.4) |
|---|---|---|---|
| **Action** | replicate hot keys across servers | migrate/swap cachelet(s) within a server | migrate/swap cachelet(s) across servers |
| **Features** | fine-grained (object/kv-pair)<br>proportional sampling<br>temporary (lease-based) | coarse-grained (cachelet/partition)<br>integer linear programming (ILP)<br>temporary (lease-based) | coarse-grained (cachelet/partition)<br>integer linear programming (ILP)<br>permanent |
| **Benefit** | **fast fix** for a few hot keys | **fast fix** for hot cachelet(s) | global load balancing |
| **Limitations** | home server bottleneck (for hot key writes)<br>scalability for large hotspots | local optimization | resource consumption<br>convergence time |
| **Cost** | **medium**<br>extra metadata (key-level)<br>extra space (duplicates) | **low**<br>extra metadata (local cachelet-level) | **high**<br>extra metadata (global cachelet-level)<br>cross-server bulk data transfer |

**Table 2:** Summary of load balancing phases of MBal. Note that cost levels (low/medium/high) represent relative costs with respect to space/performance overhead.

writes) via sampling; and (2) cachelet popularity through access rates. These statistics are collected periodically (using configurable epochs) and are used to perform an online cost/benefit analysis to trigger appropriate load balancing phases. Rebalancing is triggered only if the imbalance persists across a configurable epoch dependent number, four in our implementation, of consecutive epochs. This helps to prevent unnecessary load balancing activity while allowing MBal to adapt to workload behavior shifts.

The collected information is then used to reason about the following key design questions for each load balancing phase of MBal: (1) Why is the phase necessary? (2) When is the phase triggered and what operations are undertaken? (3) What are the costs and benefits of the phase?

### 3.2 Phase 1: Key Replication

Replication of key/object offers a fine-grained mechanism to mitigate load imbalance caused by a few extremely hot keys. This is a quick fix for short ephemeral hotspots without requiring expensive cross-server data movement.

Our key replication implementation is derived from mechanisms used in SPORE [23]. Specifically, we develop our proportional sampling technique for tracking hot keys at worker granularity based on SPORE's use of access frequency and recency. Each worker has a list of other servers (and hence other workers) in the MBal cluster. A worker with a hot key (home worker) randomly selects another server as a shadow server, and replicates the hot key to one of the associated workers on the shadow server. Depending on the hotness of a key, multiple replicas of the key can be created to multiple shadow servers. Since the replicated keys do not belong to any of the cachelets of the associated shadow servers' workers, these workers index the replicated keys using a separate (small) replica hash table. Using a separate hash table also enables shadow workers to exclude the replicated keys from being further replicated.

Upon accessing a replicated key at the home worker, a client is informed about the location of the replicas. The client can then choose any one of the replicas, which will then be used to handle *all* of the client's future read requests for that key. Writes are always performed at the home worker. Similarly as in SPORE [23], we support both synchronous and asynchronous updates. Synchronous updates have a performance overhead in the critical path, while asynchronous updates offer only eventual consistency that may result in stale reads for some clients. We leave the selection to the users based on their application needs. Furthermore, each replicated key is associated with a lease that can be renewed if the key continues to be hot or retired automatically on lease expiration. Thus, the key replication phase provides only temporary mitigation of a hotspot as the home workers for keys remain fixed.

While key replication is effective in handling short ephemeral hotspots consisting of a few keys, the approach requires both clients and MBal workers to maintain extra state of replicated keys (key locations, leases etc.). Replication also entails the using expensive DRAM for storing duplicate data. The approach also does not alleviate write-hot key based hotspots in write-intensive workloads [46]. This is because all writes are performed through a key's home worker making the home a bottleneck. To handle these limitations, multi-phased load balancer in MBal triggers other phases if the number of replicated hot keys crosses $REPL_{high}$ or if hotspots due to write-intensive workloads are observed.

### 3.3 Phase 2: Server-Local Cachelet Migration

Mitigating hotspots that consists of a large number of hot keys spanning one or more cachelets entails redistribution of the cachelets across different workers. Phase 2 represents first of the two phases of MBal, which involve such migration of cachelets. In this phase, MBal attempts to handle load imbalance at a server by triggering redistribution of cachelets to other workers running locally within the server.

As shown by the state machine in Figure 4, Phase 2 is triggered when there is high load imbalance between local workers (measured by absolute deviation (*dev*)) i.e., there exists idle or lightly-loaded local workers that can accommodate more load by swapping or migrating cachelets from the overloaded workers. MBal uses Algorithm 1 to bring down the load on the overloaded workers within an acceptable range, while not overwhelming the other lightly-loaded workers. The algorithm uses Integer Linear Pro-

**Algorithm 1:** Server-local cachelet migration.

```
Input: n_o: number of overloaded workers,
       n_t: total number of local workers,
       Set_src: set of source workers (local),
       Set_dest: set of destination workers (local),
       SERVER_LOAD_thresh: server overload threshold (e.g., 75%)
Output: Cachelet migration schedule {m}
begin
    iter ← 0
    while True do
        if n_o/n_t > SERVER_LOAD_thresh then
            trigger Phase3
            return NULL
        if n_o == 1 then
            {m} ←SolveLP1(Set_src)
        else if n_o ≥ 2 then
            {m} ←SolveLP2(Set_src, Set_dest)
        iter ← iter + 1
        if {m} == NULL then
            iter == MAX_ITER ? return Greedy(Set_src,Set_dest) : continue
        else
            break
    return {m}
```

| Notation | Description |
|---|---|
| $X_{ij}^{k}$ | 1 if cachelet $k$ is migrated from worker $i$ to worker $j$, 0 otherwise |
| $T_j$ | maximum permissible load on worker $j$[2] |
| $L_j^i$ | load of cachelet $i$ on worker $j$ |
| $L_{*j}$ | total load on worker $j$ |
| $L_{avg}$ | average load of all workers |
| $M_j^i$ | memory consumed by cachelet $i$ on worker $j$ |
| $M_{*j}$ | total memory consumed by worker $j$ |
| $M_j$ | total memory capacity of worker $j$ |
| $S$ | set of workers involved in Phase 2 |
| $S_{dest}$ | set of workers of destination cache involved in Phase 3 |
| $N$ | number of workers |

**Table 3:** Notations for the ILP model used in Phase 2 and Phase 3 of MBal load balancer.

gramming (ILP) to compute an optimal migration schedule. We also ensure that the server itself is not overloaded ($SERVER_LOAD_{thresh}$ is exceeded) before triggering Phase 2, and if so, Phase 3 is triggered instead. If ILP is not able to converge, a simple greedy algorithm will be executed eventually to reduce the *dev*.

We define two different objective functions in our linear programming model for Phase 2. The goal of the first function is to minimize the number of migration operations with a fixed source (home) worker, while that of the second function is to minimize the load deviation across all workers.

Specifically, objective (1) is to

[2] We compute this experimentally based on Amazon EC2 instance type, and the value is the same for all workers across servers.

$$minimize \quad \sum_i \sum_j X_{ij}^k \tag{1}$$

$$s.t. \quad L_{*a} + \sum_i \sum_k X_{ia}^k L_i^k - \sum_i \sum_k X_{ai}^k L_a^k \le T_a \tag{2}$$

$$\forall i \in S \setminus a : L_{*i} + \sum_k X_{ai}^k L_a^k - \sum_k X_{ia}^k L_i^k \le T_i \tag{3}$$

where $a$ is the index of the fixed source worker thread. Objective (2) is to

$$minimize \quad \frac{\sum_{i \in S} \left| L_{*i} + \sum_{j \in S \setminus i} \sum_k X_{ji}^k L_j^k - \sum_{j \in S \setminus i} \sum_k X_{ij}^k L_i^k - L_{avg} \right|}{N} \tag{4}$$

$$s.t. \quad \forall i \in S : L_{*i} + \sum_{j \in S \setminus i} \sum_k X_{ji}^k L_j^k - \sum_{j \in S \setminus i} \sum_k X_{ij}^k L_i^k \le T_i \tag{5}$$

Both objectives are subject to the following constraints:

$$\forall i, \forall j \in S : X_{ij} = 0 \text{ or } 1 \ , \tag{6}$$

$$\forall i \in S : 0 \le \sum_{j \in S \setminus i} X_{ij} \le 1 \ . \tag{7}$$

Table 3 describes the notations used for representing the above models. Constraints 2 and 3 are used to restrict the load after migration on the source and destination workers under a permissible limit ($T_j$), respectively. Similarly, constraint 5 restricts the load for each worker involved in optimizing objective (2). Both objectives share constraints 6 and 7, implying that migration decisions are binary and a cachelet on a particular source worker can only be migrated to one destination worker. Objective (1) is used when only a single worker is overloaded on a MBal server, and objective (2) is used for all other cases. Given the long computation time required by objective (2) to finish, we relax the objective by dividing the optimization phase into multiple iterations. In each iteration, the algorithm picks at most two overloaded threads as sources and two lightly-loaded ones as destinations. This heuristic shortens the search space for ILP, ensuring it does not affect overall cache performance and still provide good load balance (shown in §4.2).

MBal adopts a seamless server-local cachelet migration mechanism that incurs *near-zero* cost. Since each cache server partitions resources into cachelets that are explicitly managed by dedicated worker threads, server-local migration from one worker to another requires no data copying or transfer overheads (all workers are part of the same single address space). Only the owner worker of the cachelet is changed. Similarly as in Phase 1, we use a lease-based timeout for migrated cachelets, which means that a migrated cachelet is returned to its original home worker when the associated hotspot no longer persists. This helps address ephemeral hotspots, while allowing for the cachelets to be restored to their home workers with negligible overhead.

Clients are informed of migrated cachelets whenever the home worker receives any requests about the cachelets. Since we use leases, clients cache the home worker information and update the associated mappings to the new worker. Using the cached home worker information, clients can restore the original mapping after the lease expires.

While Phase 2 provides a lightweight mechanism to mitigate load imbalance within a server, its utility is limited by the very nature of its local optimization goals. For cases with server-wide hotspots and to provide long-term rebalancing of cachelets across MBal cluster, coordinated migration management is required.

### 3.4 Phase 3: Coordinated Cachelet Migration

An overloaded cache server or lack of an available local worker that can handle a migrated cachelet implies that Phase 2 cannot find a viable load balancing solution, thus Phase 3 is triggered. In MBal's Phase 3, cachelets from one cache server are offloaded to one or more lightly-loaded servers in the cluster.

**Algorithm 2:** Coordinated cachelet migration.

```
Input: Global cache server stats array: V_S,
input source worker: src,
IMB_thresh: Imbalance threshold
Output: Cachelet migration schedule V_M
begin
    iter ← 0
    while dev(LOAD(src), LOAD(S_dest)) > IMB_thresh
    && iter < MAX_ITER do
        S_dest ← min(V_S)        // get minimum loaded server
        V_temp ← SolveLP(src, S_dest)
        if V_temp == NULL then
            V_temp ← Greedy(src, S_dest)
        V_M ← V_M ∪ V_temp
        iter ← iter + 1
    if all cluster is hot or src still too hot then
        return NULL              // add new cache servers to scale out
    return V_M
```

Under Phase 3, an overloaded worker ($src$) notifies the centralized coordinator to trigger load balancing across cache servers. The coordinator periodically fetches statistics from all cluster workers including cachelet-level information about request arrival rate (load), amount of data stored (memory consumption), and read/write ratio.

Algorithm 2 then utilizes these statistics to choose a target server ($S_{dest}$) with the lowest load to redistribute cachelets from the source overloaded worker to the target's workers. The output of the algorithm is a list of migration commands each specifying a cachelet ID and the address of a corresponding destination worker thread to which the cachelet should be migrated. The commands are then executed by the workers involved in the migration.

During each iteration, the algorithm selects the most lightly-loaded destination server and creates a candidate list of cachelets to migrate using an ILP routine whose objective is to minimize the gap between the load of source and destination workers using Equation 8 as follows:

$$minimize \quad \frac{\sum_{i \in S'} \left| L_{*i} + \sum_{j \in S' \setminus i} \sum_k X_{ji}^k L_j^k - \sum_{j \in S' \setminus i} \sum_k X_{ij}^k L_i^k - L_{avg} \right|}{N} \tag{8}$$

$$s.t.$$

$$\forall i \in S' : L_{*i} + \sum_{j \in S' \setminus i} \sum_k X_{ji}^k L_j^k - \sum_{j \in S' \setminus i} \sum_k X_{ij}^k L_i^k \le T_i \tag{9}$$

$$M_{*a} + \sum_i \sum_k X_{ia}^k M_i^k - \sum_i \sum_k X_{ai}^k M_a^k \le M_a \tag{10}$$

$$\forall i \in S_{dest} : M_{*i} + \sum_k X_{ai}^k M_a^k - \sum_k X_{ia}^k M_i^k \le M_i \tag{11}$$

$$\forall i, \forall j \in S' : X_{ij} = 0 \text{ or } 1 \tag{12}$$

$$\forall i \in S' : 0 \le \sum_{j \in S' \setminus i} X_{ij} \le 1 \tag{13}$$

where $a$ is the index of the source worker thread, and $S' = S_{dest} \cup a$.

Similar to Phase 2's ILP, constraint 9 bounds the load on any worker below a pre-specified permissible limit ($T_i$). However, unlike Phase 2, data is actually transferred across servers in Phase 3 coordinated migration. Thus, we need to ensure that the destination server has enough memory capacity to hold the migrated cachelets without causing extraneous evictions. To this end, constraints 10 and 11 ensure that the memory availability of the source and destination workers is not exceeded.

Moreover, if the ILP does not converge, – in rare cases, e.g., when the choice of a destination server is restricted in each iteration, it may not be possible to satisfy both the memory and load constraints – we employ a simple greedy solution to reduce as much load as possible from the overloaded worker.

MBal maintains consistency during migration by adopting a low-overhead Write-Invalidate protocol. We employ a special migrator thread on each cache server for this purpose. Instead of migrating the entire cachelet in a single atomic operation, we migrate the tuples belonging to the selected cachelet on a per-bucket basis. We use hash table based indexes for cachelets. While keys in a bucket are being migrated, the affected worker continues serving client requests for the other buckets. Only the requests for the bucket under migration are queued. Any `UPDATE` requests to existing keys that have already been migrated result in invalidation in both the destination (home after migration) and the source (home before migration) hash table. The `INSERT` requests for new items are treated as `NULL` operations and are directly sent to backend store. This causes no correctness problems because MBal like Memcached is a read-only, write through cache and has no dirty data. The `GET` requests are serviced for the data that is valid in the source hash table. Once the migration is complete, the source worker thread informs the coordinator about the change in the cachelet mapping. In our de-

sign, the clients exchange periodic heartbeat messages with the coordinator, and any change in cachelet mapping information is communicated to the clients by the coordinator as a response to these messages. The mapping change information only needs to be stored at the coordinator for a maximum configurable period that is slightly longer than the clients' polling period. This has two benefits: (1) it guarantees that all clients will eventually see the mapping changes within a short period of time; and (2) the coordinator is essentially stateless and only has to maintain the state information for a short time (during migration of a cachelet). Hence, after all active clients have updated their mapping tables, the coordinator informs the source worker thread to delete any metadata about its ownership of the transferred cachelets.

As evident by the design discussion, Phase 3 is the most expensive load balancing mechanisms employed by MBal. It serves as a last resort and is only activated when the first two phases cannot effectively mitigate the impact of load imbalance and hotspots persist. While the coordinator does not play a role during normal operation, admittedly, it can become a bottleneck for servicing the migration requests, especially if a large number of servers in the cluster are overloaded simultaneously (implying a need for adding more servers in the cluster, which is beyond the scope of the paper). Also, a failure of the coordinator during periods of imbalance can cause hotspots to persist or cache servers to maintain migration state longer (until the coordinator is brought back up). While not part of this work, we plan to exploit numerous existing projects [25, 41] in this domain to augment our coordinator design to provide more robust fault tolerance for Phase 3.

### 3.5 Discussion

While the techniques presented in this Section are applicable in any (physical or virtualized) deployment of in-memory caching clusters, use of virtualized infrastructure in the cloud is likely to demonstrate higher load imbalance and performance fluctuations due to the factors such as resource sharing across multiple tenants, compute/memory resource oversubscription, etc. Hence, MBal focuses on providing a robust in-memory caching framework for the cloud without directly dealing with multi-tenancy and resource allocation issues, which are beyond a cloud tenant's control.

## 4. Evaluation

MBal uses a partitioned lockless hash table as its core data structure. We support the widely used Memcached protocol (with API to support `GET`, `SET`, etc.). To this end, we extend libmemcached [5] and SpyMemcached [10] to support client-side key-to-thread mapping. The central coordinator is written in Python and uses Memcached protocol (pylibmc [8]).

We present the evaluation of MBal using a local testbed and a 20-node cluster on Amazon EC2 (`c3.large`). We evaluate the cache performance on 8-core and 32-core physical commodity servers, examine individual phases of MBal, and finally study the end-to-end system performance on a cloud cluster.

### 4.1 MBal Performance: Normal Operation

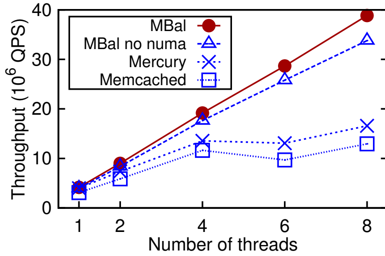
In our first set of experiments, we evaluate our design choices in building a lockless key-value cache (§2). We also compare the performance of MBal with existing systems such as Memcached ($v$1.4.19), Mercury [21] and their variants. Unless otherwise stated, we perform all tests on a dual-socket, 8-core 2.5 GHz machine with 10 Gbps Ethernet and 64 GB DRAM. Also, we enabled 2 MB hugepage support in our tests to reduce TLB misses.

***Microbenchmark Performance*** To highlight the bottlenecks inherent in the designs of different key-value caches, we perform our next tests on a single machine. Here, each worker thread generates its own load, and there are no remote clients and thus no network traffic. First, we use two workloads, one with GET-only and the other with SET-only requests. We use a 5 GB cache, the size of which is larger than the working set of the workloads ($\approx$ 1150 MB), thus avoiding cache replacement and its effects on the results. Each workload performs a total of 32 million operations using fixed-size key-value pairs (10 B keys and 20 B values)[3]. For the GET workload, we pre-load the cache with 40 million key-value pairs, while the SET workload operates without pre-loading. The key access probability is uniformly distributed in both workloads. For fair comparison, we set the thread-local memory buffer to be the same size (256 MB) in both MBal and Mercury.

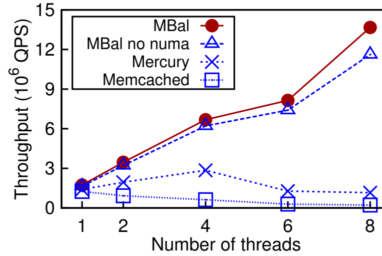
Figure 5(a) and Figure 5(b) show the throughput in terms of $10^6$ QPS (MQPS) for the three systems. We observe that MBal's performance scales as the number of threads increases for both GET and SET workloads. Mercury, which uses fine-grained bucket locking, performs better than Memcached that suffers from synchronization overheads due to its coarse-grained locks. However, Mercury is not able to match the throughput of MBal due to MBal's *end-to-end* lockless design that removes most synchronization bottlenecks and allows independent resource management for each thread. Thus, for the GET workload with six threads, MBal is able to service about 2.3× more queries than Mercury, as threads in Mercury still contend for bucket-level locks.

In case of SET operations, whenever a key-value pair is overwritten, the old memory object has to be freed (pushed to a free pool). Under MBal, we garbage-collect this memory back to the thread-local free memory pool (recall that MBal transfers memory from thread-local to global free pool in bulk only under the conditions discussed in §2.4). In contrast, Mercury pushes the freed memory back into the

[3] Our experiments with different (larger/smaller) key and value sizes show similar trends, hence are omitted due to space constraints.




(a) GET performance.




(b) SET performance.

**Figure 5:** Microbenchmark performance.

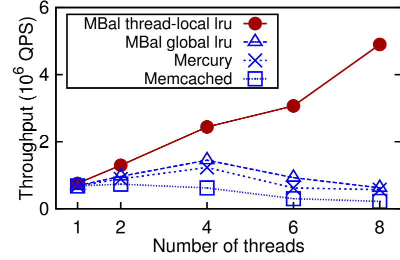
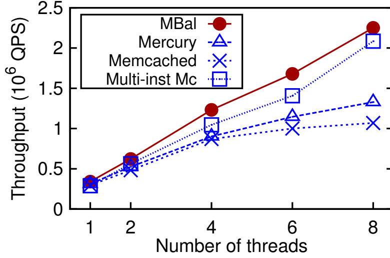
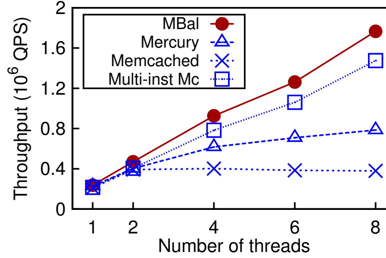


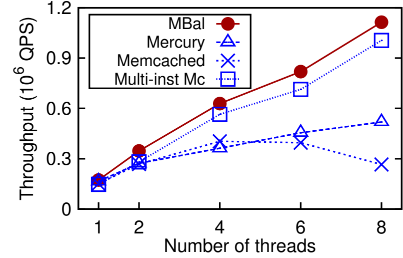

**Figure 6:** 15% cache miss.




(a) 95% GET.




(b) 75% GET.




(c) 50% GET.

**Figure 7:** Complete MBal system performance under varying GET/SET ratios. For multi-instance Memcached (`Multi-inst Mc`), the number of threads represents number of instances, i.e., there is one worker thread per instance.

global memory pool similarly as in Memcached. This introduces another synchronization overhead for write-dominant workloads in addition to the lock contention on the hash table. Thus, by mitigating both of these aspects, MBal is able to provide about 12× more throughput on the insert path for the eight workers case. In order to evaluate the impact of NUMA-based multi socket machines on cache performance, we also perform experiments on MBal with NUMA-awareness disabled. Under an 8-thread setup, MBal with NUMA-awareness achieves about 15% and 18% higher throughput for GET and SET operations, respectively, compared to MBal with no NUMA support (`MBal no numa`). The scaling trends and the widening performance gap between the studied systems as concurrency is increased (Figure 5) shows the benefits of MBal's lockless design and memory management policies.

Finally, we run a write-intensive workload on a 1 GB cache that is smaller than the working set, where about 15% GETs miss in the cache. Each miss triggers a SET to insert. Figure 6 shows that MBal with thread-local memory pools achieves about 5 MQPS. On the other hand, MBal with only global memory pool (`MBal global lru`) achieves similar performance to Mercury and Memcached, i.e., 0.5 MQPS, which is about an order of magnitude lower than MBal with thread-local pools (`MBal thread-local lru`).

***Complete Cache System Performance*** To test end-to-end client/server performance under MBal, we use a setup of five machines (1 server, 4 clients) with the same configuration as in §4.1. To amortize network overheads, we use `MultiGET` by batching 100 GETs. We use workloads generated by YCSB [15] using varying GET/SET ratios. Each workload consists of 8 million unique key-value pairs (10 B key and 20 B value) with 16 million operations generated using Zipfian key popularity distribution (Zipfian constant: 0.99). This setup helps us simulate real-world scenarios with skewed access patterns [12]. Each worker maintains 16 cachelets.

As shown in Figure 7, not only does MBal's performance scale with the number of worker threads for read-intensive workloads, it is also able to scale performance across workloads that are write intensive. For example, for a workload with 25% writes (Figure 7(b)), MBal with 8 threads outperforms both Memcached and Mercury by a factor of 4.7 and 2.3, respectively. MBal can scale up to the number of cores in the system (8 cores) for all workloads, while Memcached fails to scale with increased concurrency. This shows that as interconnects become faster, the design choices of Memcached will start to affect overall performance and MBal offers a viable alternative.

***Impact of Dynamic Memory Allocation*** During our tests, we found that for Memcached to scale, we need to run multiple single-threaded instances (`Multi-inst Mc`). However, in our opinion, not only is such a deployment/design qualitatively inferior (§2.5), but as shown in Figure 8, it incurs significant overhead. For example, 8-instance `Multi-inst Mc(malloc)` achieves 8% less QPS on average (with value sizes ranging from 32 B to 1024 B) compared to `Multi-inst Mc(static)` due to the overhead of `malloc`. This overhead increases to 13% when we replace our optimized MBal slab

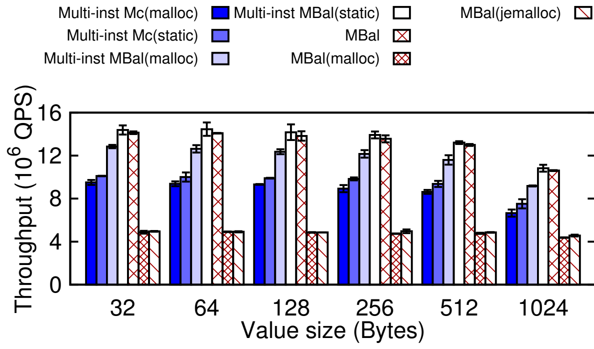



**Figure 8:** Impact of dynamic memory allocator on performance of 8 instance/thread cache. We run 100% SET workload with varying value sizes. We use `glibc`'s `malloc` and `jemalloc` as an option for cache slab allocation.

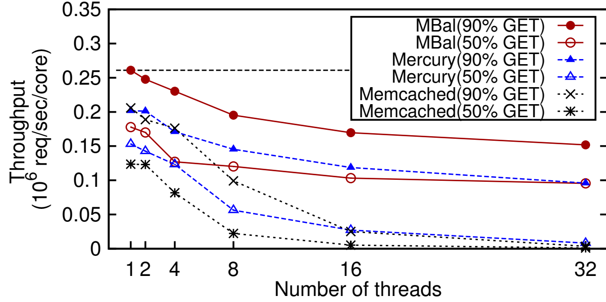



**Figure 9:** MBal scalability on a dual-socket, 32-core, 2 GHz machine. Workloads (16 B key, 32 B value) are generated from three 32-core machines in a LAN with 10 GbE interconnect, using `memaslap` with `MultiGET` enabled. Each workload runs for 60 seconds. The Y-axis shows per-core throughput; note the horizontal line shows ideal scalability.

allocator with `malloc`. For multi-threaded MBal, `jemalloc` does not scale due to lock contention.

***Scalability on Many-core Machine*** Figure 9 demonstrates the scalability of MBal on a many-core machine. For both read-intensive (90% GET) and write-intensive (50% GET) workloads MBal achieves 18.6× and 17.2× the one-core performance, respectively, for 32 cores. The factors that limit the single-machine scalability are kernel packet processing and network interface interrupt handling. This is observed as a large fraction of CPU cycles being spent in `system mode` and servicing of `soft IRQs`[4]. As observed, both Memcached and Mercury do not scale well for write-intensive workload due to cache lock contention.

[4] There are a number of existing works [28, 29] that we can leverage to further improve the multi-core scalability.

## 4.2 MBal Performance: Load Balancer

***Experimental Setup*** We perform our next set of tests on a real Amazon EC2 cluster with 20 `c3.large` instances acting as cache servers. Clients are run on a separate cluster with up to 36 `c3.2xlarge` instances—that are in the same availability zone `us-west-2b`—as the cache servers. We provision both the client and server cluster instances on shared-tenant hardware. Similarly, the central coordinator of MBal's Phase 3 is run on a separate `c3.2xlarge` instance, also in the same availability zone as the servers and clients.

### 4.2.1 Performance of Individual Phases

***Workloads*** The workloads are generated using YCSB and consists of 20 million unique key-value tuples (24 B keys and 64 B values). Each client process generates 10 million operations using the Zipfian distribution (Zipfian constant = 0.99). The workload is read-intensive with 95% GETs and 5% SETs. We run one YCSB process using 16 threads per client, and then increase the number of clients until the cache servers are fully saturated.

***Phase 1: Key Replication*** Here, we only enable Phase 1 of MBal, and use a key sampling rate of 5%. Figure 10 depicts the average $99^{th}$ percentile read tail latency and aggregated throughput trade-off observed under our workload on different system setups. `Memcached`, `Mercury`, and `MBal (w/o load balancer)` represent the three setups of Memcached, Mercury, and MBal, respectively. `MBal (Unif)` shows the scenario with uniform workload (under MBal) and provides an upper bound for the studied metrics. We observe that without key replication support, MBal achieves only about 5% and 2% higher maximum throughput compared to the corresponding case with the same number of clients for Memcached and Mercury, respectively. This is because scaling-out the caches with associated virtualization overhead diminishes the benefits of vertical (scale-up) scalability on each individual server. However, when hot keys are replicated to one shadow server in `MBal (P1)` (key replica count = 2), the maximum throughput is observed to improve by 17%, and the $99^{th}$ percentile latency by 24% compared to the case with no replication. Thus, MBal is able to effectively offload the heat for hot keys, and mitigate the performance bottleneck observed in Memcached and Mercury.

***Phase 2: Server-Local Cachelet Migration*** Next, we only turn on the Phase 2 of MBal and study its performance under our workloads. In Figure 10 we observe that, compared to the baseline MBal without server-local cachelet migration, Phase 2 achieves 8% higher maximum throughput and 14% lower tail latency. By migrating entire cachelets to less loaded threads, we are better able to mitigate the skew in the load. This not only helps to increase the throughput, but also improves overall latency characteristics as well. Moreover, our design ensures that the migration is effectively achieved by a simple modification of a pointer within the server, and thus incur little overhead.

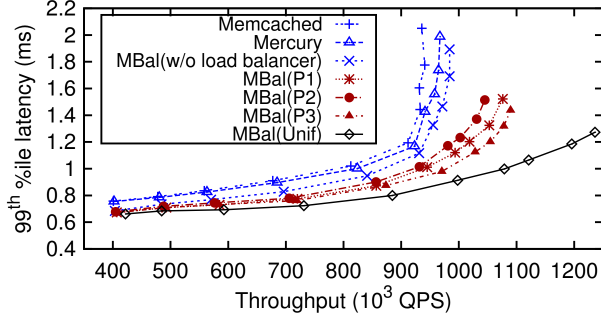



**Figure 10:** The $99^{th}$ percentile latency and aggregated throughput achieved by key replication (P1), server-local cachelet migration (P2), and coordinated cachelet migration (P3). We vary the number of clients from 10 to 34 to increase the throughput (shown on the X-axis). `Unif` represents uniform workload.

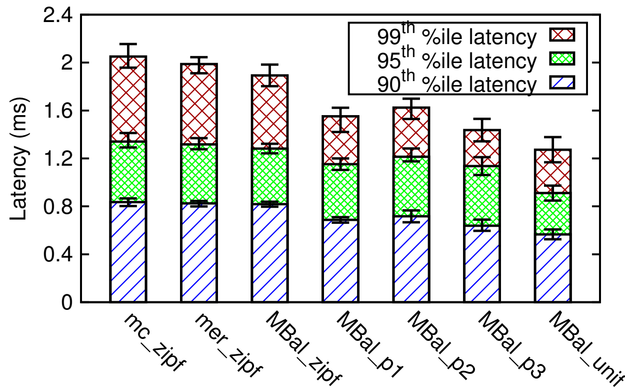



**Figure 11:** Breakdown of latency achieved under different configurations. `MBal_p1`, `MBal_p2`, and `MBal_p3` represents MBal with only Phase 1, Phase 2, and Phase 3 enabled, respectively. `mc` represents Memcached, and `mer` represents Mercury.

***Phase 3: Coordinated Cachelet Migration*** Figure 10 shows the results with only Phase 3 enabled. We observe an improvement in the maximum throughput of up to 20% and 14%, compared to Memcached and `MBal (w/o load balancer)`, respectively. Coordinated cachelet migration also decreases the average $99^{th}$ percentile tail read latency by 30% and 24% compared to Memcached and `MBal (w/o load balancer)`, respectively. On the flip side, the migration incurs a high cost. We observed that migrating one cachelet at MBal cache server's peak load takes 5 to 6 seconds on average. This is the root cause of the long convergence time of Phase 3. Moreover, the CPU utilization on the central coordinator was observed to be 100% when doing the ILP computation. This shows that migration is an effective way to mitigate load imbalance. However, the increasing traffic due to migration and the increased CPU load on the centralized coordinator suggest that the approach should only be adopted for sustained hotspots and that too as a last resort.

| Workload | Characteristics | Application Scenario |
|---|---|---|
| `WorkloadA` | 100% read, Zipfian | User account status info |
| `WorkloadB` | 95% read, 5% update, hotspot (95% ops in 5% data) | Photo tagging |
| `WorkloadC` | 50% read, 50% update, Zipfian | Session store recording recent actions |

**Table 4:** Workload characteristics and application scenarios used for testing the multi-phase operations of MBal.

As part of our future work, we are exploring techniques for developing a hierarchical/distributed load balancer to reduce the cost of such migration.

***Trade-off Analysis*** We have seen that for the same workload, each of the three phases of MBal are able to improve the throughput and tail latency to some extent. In our next experiment, we study the trade-offs between the different phases. Figure 11 plots the breakdown of read latency experienced under different cache configurations. The key replication of Phase 1 provides randomized load balancing by especially focusing on read-intensive workloads with Zipfian-like key popularity distributions. Phase 2's server-local cachelet migration serves as a lightweight intermediate stage that offers a temporary fix in response to changing workload. The limitation of Phase 2 is that it cannot offload the "heat" of overloaded cache servers to a remote server that has spare capacity. This is why the performance improvement under Phase 2 is slightly less than that under Phase 1. For instance, Phase 1 is 4.2%, 5.1% and 4.4% better for $90^{th}$, $95^{th}$ and $99^{th}$ percentile latency than Phase 2, respectively, as shown in Figure 11. Phase 3 relies on heavyweight coordinated cachelet migration—if other phases cannot achieve the desired load balance—which can optimally re-distribute load across the whole cluster and provides a better solution than the randomized key replication scheme.

### 4.2.2 Putting It All Together

In our next experiment, we evaluate the adaptivity and versatility of MBal with all three phases enabled. We use a dynamically changing workload for this test, which is generated by YCSB and is composed of three sub-workloads shown in Table 4. The sub-workloads are provided by the YCSB package for simulating different application scenarios. Note that, we adjust `WorkloadB` to use YCSB's hotspot key popularity distribution generator instead of the original Zipfian distribution generator. These workloads also resemble characteristics of Facebook's workloads [46].

Each sub-workload runs for 200 seconds and then switches to the next one. To get an upper-bound on performance, we also run the three workloads under uniform load distribution. We conduct two baseline runs, one under Memcached and the other under MBal with load balancer disabled. To quantitatively understand how each single phase reacts under the different workload characteristics of this test, we first perform three tests, each with one single phase enabled (simi-

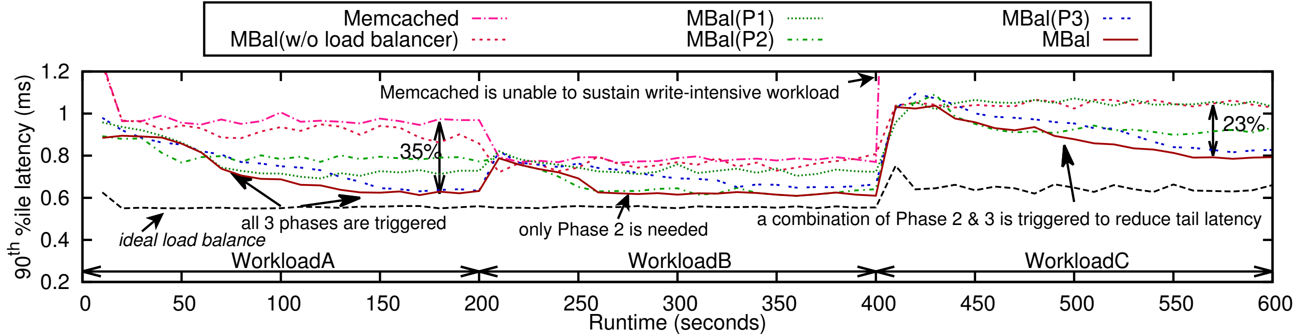



**Figure 12:** $90^{th}$ percentile read latency timeline for a dynamically changing workload. The overall workload is a combination of `WorkloadA`, `WorkloadB` and `WorkloadC`. Spikes at around $200$ $s$ and $400$ $s$ occur because of sudden workload shifts.

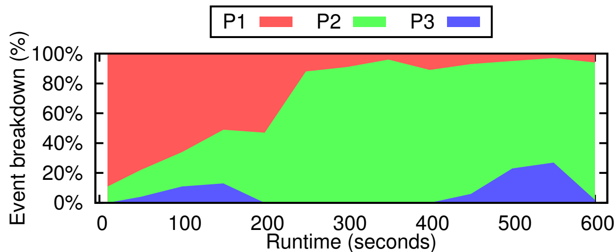



**Figure 13:** Breakdown of phase triggering events observed under the multi-phase test.

larly as for individual phase tests). Then we perform a fourth run with all three phases enabled to study the end-to-end behavior of MBal. Figure 12 plots the $90^{th}$ percentile read tail latency change[5]. Next, Figure 13 depicts the breakdown of phase triggering events that corresponds to the execution of the multi-phase test of Figure 12. We see that MBal reacts quickly to workload changes and employs an appropriate load balancing strategy accordingly.

***Adaptivity and Versatility*** Under `WorkloadA`, Phase 1 and Phase 2 stabilize after around $70$ $s$ and $50$ $s$, respectively, while it takes longer, i.e., around $150$ $s$, for Phase 3 to stabilize the latency. Phase 3 eventually achieves slightly lower latency, compared to Phase 1, though only a limited number of cachelets are migrated. This is because, as observed before, Phase 3's optimal migration solutions perform better than randomized key replication of Phase 1. With all three phases combined, the clients see steady reduction of latency in a more smooth fashion. This is mainly due to Phase 2 serving as an effective backup approach for cache servers where key replication cannot gracefully bring the load back down to normal. As observed in Figure 13, Phase 3 is eventually triggered on a small number ($\approx$ $12\%$ of all the triggered events) of servers where all worker threads are overloaded. Thus, the impact of the overhead of Phase 3 is reduced by the use of other phases. These results demonstrate the effectiveness of a multi-phase load balancing mechanism where the phases complement each other.

`WorkloadB` begins at $200$ $s$. At this time, Phase 2's scheme immediately starts to re-balance the load. Note that Phase 1's effectiveness dramatically diminishes during this sub-workload, since hotspot distribution generator uniformly generates requests that are concentrated in $5\%$ of all tuples. The effect of this is that the load distribution across the cache cluster is semi-uniform, whereas within each server, worker threads see non-uniformly distributed load. Phase 2 captures this behavior and promptly adapts the latency accordingly. Note that, if Phase 3 were the only available approach, it would eventually improve latency, but with a significantly long convergence duration. However, in this case, Phase 2 is triggered and ends up adapting the latency throughout the duration of `WorkloadB`. This result demonstrates that not only can Phase 2 serve as an intermediate and complementary phase for smoothing out latency variations, it can also serve as the main load balancing solution when necessary under some scenarios.

`WorkloadC` is a write-heavy sub-workload that starts at $400$ $s$. Once again, Phase 1 is unable to detect hotspots as its key tracking counter uses weighted increments on read and weighted decrements on writes. This is because otherwise the overhead of propagating the writes to the key replicas would outweigh the benefits of load balancing. Here, Phase 2 can in itself effectively lower down the latency to some extent. However, Phase 3 kicks in for some of the servers to ensure that the system as a whole does not suffer from load imbalance. Thus, MBal is able to achieve its overall load-balancing goals.

Figure 13 shows that, unlike Phase 1 and Phase 2 that are actively invoked to balance the load throughout the three workloads, only $13\%$ (on average) of the load balancing events involve Phase 3. This further demonstrates that Phase 3 is only sparingly used and thus is not the bottleneck in our load balancing framework.

[5] We observed a similar trend for write latency.

## 5. Related Work

***High Performance In-Memory Key-Value Stores*** Improving performance of in-memory key-value storage systems is the focus of much recent research [19, 35, 37, 45]. Specifically, systems such as Memcached on Tilera [13], Chronos [30] and MICA [36] use exclusively accessed per-core partitions to eliminate global lock contention. Similarly, MBal exploits per-core partitioning to improve performance. MBal, in addition, provides load balancing across servers as well as the ability to scale-up to fully exploit the multi-core architecture.

***Storage/Memory Load Balancing*** Distributed hash tables (DHT) have been extensively used to lower the bandwidth consumption of routing messages while achieving storage load balancing in peer-to-peer networks. Virtual server [17] based approaches [22, 32, 43] have also been studied in this context. MBal differs from these works in that it focuses on adaptive and fast-reactive access load balancing for cloud-scale web workloads.

Proteus [34] is a dynamic server provisioning framework for memory cache cluster, which provides deterministic memory load balancing under provisioning dynamics. Similarly, Hwang et al. [27] proposed an adaptive hash space partitioning approach that allows hash space boundary shifts between unbalanced cache nodes without further dividing the hash space. The associated framework relies on a centralized proxy to dispatch all requests from the web servers; and the centralized load balancer is actively involved in transferring data from old cache nodes to new ones. In contrast, MBal considers the memory utilization for cross-server migration for access load balancing. MBal also has the benefit of avoiding a centralized component that is inline with the migration; the centralized coordinator of MBal is used for directing only global load balancing when needed.

***Access Load Balancing*** Replication is an effective way for achieving access load balancing. Distributed file systems such as Hadoop Distributed File System (HDFS) [44] place block replicas strategically for fault tolerance, better resource efficiency and utilization. At a finer granularity, SPORE [23] uses an adaptive key replication mechanism to redistribute the "heat" on hot objects to one or more shadow cache servers for mitigating the queuing effect. However, workloads can develop sustained and expanding unpredictable hotspots (i.e., hot shards/partitions) [24], which increase the overhead of maintaining key-level metadata on both the client and server side. In contrast, MBal is effective in handling such load imbalance as it employs a multi-phase adaptation mechanism with different cost-benefits at different levels.

Research has looked at handling load imbalance on the caching servers by caching a small set of extremely popular keys at a high-performance front-end server [20]. While Fan et al. [20] focus on achieving load balancing for an array of wimpy nodes by caching at a front-end server, Zhang et al. [47] propose hotspot redirection/replication using a centralized high-performance proxy placed in front of a cluster of heterogeneous cache servers. MBal tries to handle load imbalance within the existing caching servers without introducing another layer of indirection or other centralized bottlenecks. The centralized coordinator in MBal is sparingly used only when other phases are not sufficient to handle the hotspots. In our future work, we plan to investigate the use of distributed/hierarchical coordinators to further reduce the bottleneck of our existing coordinator if any.

Chronos [30] uses a greedy algorithm to dynamically reassign partitions from overloaded threads to lightly-loaded ones to reduce Memcached's mean and tail latency. Similar to Chronos, MBal also adopts a partition remapping scheme as a temporary fix to load imbalance within a server. However, MBal has a wider scope in handling load imbalance, covering both a single server locally, as well as globally across the whole cache cluster.

Memcached community has implemented virtual buckets [9] as a library for supporting replication and online migration of data partitions when scaling out the cache cluster. Couchbase [16] uses this mechanism for smoothing warm-up transitioning and rebalancing the load. MBal uses a similar client-side hierarchical mapping scheme to achieve client-side key-to-server remapping. However, MBal differs from such work in that it uses cachelet migration across servers as a last resort only, and preserves the distributed approach of the original Memcached except in the small number of cases when the global rebalancing is necessitated.

## 6. Conclusion

We have presented the design of an in-memory caching tier, MBal, which adopts a fine-grained, horizontal per-core partitioning mechanism to eliminate lock contention, thus improving cache performance. It also cohesively employs different migration and replication techniques to improve performance by load balancing both within a server and across servers to re-distribute and mitigate hotspots. Evaluation for single-server case showed that MBal's cache design achieves $12\times$ higher throughput compared to a highly-optimized Memcached design (Mercury). Testing on a cloud-based 20-node cluster demonstrates that each of the considered load balancing techniques effectively complement each other, and compared to Memcached can improve latency and throughput by $35\%$ and $20\%$, respectively.


## Acknowledgements

We are grateful to the anonymous reviewers and our shepherd, Thomas Moscibroda, for their valuable feedback, which significantly improved the paper. We also thank Ali Anwar and M. Safdar Iqbal for manuscript proofreading that helped improve the readability. This work was sponsored in part by the NSF under CNS-1405697 and CNS-1422788 grants.

## References


[1] Amazon EC2 Pricing. `http://aws.amazon.com/ec2/pricing/`.

[2] Amazon Web Service ElastiCache. `http://aws.amazon.com/elasticache/`.

[3] Cache with Twemcache. `https://blog.twitter.com/2012/caching-with-twemcache`.

[4] How Twitter Uses Redis to Scale. `http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale.html`.

[5] libmemcached. `http://libmemcached.org/`.

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

[7] Memcached Protocol. `https://code.google.com/p/memcached/wiki/NewProtocols`.

[8] pylibmc. `https://pypi.python.org/pypi/pylibmc`.

[9] Scaling Memcached with vBuckets. `http://dustin.sallings.org/2010/06/29/memcached-vbuckets.html`.

[10] SpyMemcached. `https://code.google.com/p/spymemcached/`.

[11] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91, New York, NY, USA, 1998. ACM.

[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[13] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, July 2011.

[14] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[16] Couchbase. vbuckets: The core enabling mechanism for couchbase server data distribution (aka "auto-sharding"). Technical Whitepaper.

[17] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 202–215, New York, NY, USA, 2001. ACM.

[18] J. Evans. A scalable concurrent malloc(3) implementation for freebsd. In *BSDCAN'06*, 2006.

[19] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[20] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[21] R. Gandhi, A. Gupta, A. Povzner, W. Belluomini, and T. Kaldewey. Mercury: Bringing efficiency to key-value stores. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 6:1–6:6, New York, NY, USA, 2013. ACM.

[22] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2253–2262 vol.4, March 2004.

[23] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 13:1–13:17, New York, NY, USA, 2013. ACM.

[24] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 8:1–8:7, New York, NY, USA, 2014. ACM.

[25] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[26] J. Hwang, K. K. Ramakrishnan, and T. Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, Apr. 2014. USENIX Association.

[27] J. Hwang and T. Wood. Adaptive performance-aware distributed memory caching. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 33–43, San Jose, CA, 2013. USENIX.

[28] Intel Corporation. Intel data plane development kit: Getting started guide.

[29] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, Apr. 2014. USENIX Association.

[30] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud*

*Computing*, SoCC '12, pages 9:1–9:14, New York, NY, USA, 2012. ACM.

[31] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM STOC '97*, 1997.

[32] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 36–43, New York, NY, USA, 2004. ACM.

[33] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[34] S. Li, S. Wang, F. Yang, S. Hu, F. Saremi, and T. Abdelzaher. Proteus: Power proportional memory cache cluster in data centers. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 73–82, 2013.

[35] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 27:1–27:14, New York, NY, USA, 2014. ACM.

[36] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.

[37] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[38] T. Marian, K. S. Lee, and H. Weatherspoon. Netslices: Scalable multi-core packet processing in user-space. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 27–38, New York, NY, USA, 2012. ACM.

[39] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, Apr. 2014. USENIX Association.

[40] 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 *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[41] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[42] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 12:1–12:14, Berkeley, CA, USA, 2008. USENIX Association.

[43] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *IPTPS'03*, 2003.

[44] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[45] A. Wiggins and J. Langston. Enhancing the scalability of memcached, 2012.

[46] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Characterizing facebook's memcached workload. *Internet Computing, IEEE*, 18(2):41–49, Mar 2014.

[47] W. Zhang, J. Hwang, T. Wood, K. Ramakrishnan, and H. Huang. Load balancing of heterogeneous workloads in memcached clusters. In *9th International Workshop on Feedback Computing (Feedback Computing 14)*, Philadelphia, PA, June 2014. USENIX Association.