Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System

Jiacheng Shen[†1], Pengfei Zuo[§], Xuchuan Luo[‡1], Yuxin Su[¶],
Jiazhen Gu[†], Hao Feng[§], Yangfan Zhou[‡], Michael R. Lyu[†]
[†]The Chinese University of Hong Kong, [§]Huawei Cloud, [‡]Fudan University, [¶]Sun Yat-sen University

Abstract

In-memory caching systems are fundamental building blocks in cloud services. However, due to the coupled CPU and memory on monolithic servers, existing caching systems cannot elastically adjust resources in a resource-efficient and agile manner. To achieve better elasticity, we propose to port in-memory caching systems to the disaggregated memory (DM) architecture, where compute and memory resources are decoupled and can be allocated flexibly. However, constructing an elastic caching system on DM is challenging since accessing cached objects with CPU-bypass remote memory accesses hinders the execution of caching algorithms. Moreover, the elastic changes of compute and memory resources on DM affect the access patterns of cached data, compromising the hit rates of caching algorithms. We design Ditto, the first caching system on DM, to address these challenges. Ditto first proposes a *client-centric caching framework* to efficiently execute various caching algorithms in the compute pool of DM, relying only on remote memory accesses. Then, Ditto employs a *distributed adaptive caching scheme* that adaptively switches to the best-fit caching algorithm in real-time based on the performance of multiple caching algorithms to improve cache hit rates. Our experiments show that Ditto effectively adapts to the changing resources on DM and outperforms the state-of-the-art caching systems by up to 3.6× in real-world workloads and 9× in YCSB benchmarks.

CCS Concepts: • **Information systems** → **Distributed storage**.

Keywords: Disaggregated Memory, RDMA, Key-Value Cache

1 Introduction

In-memory caching systems, *e.g.*, Memcached [48] and Redis [58], are widely adopted in cloud services [9, 16, 71, 84] to reduce service latency and improve throughput. Due to the dynamic and bursty characteristics of requests in cloud services [62, 69, 82], elasticity, *i.e.*, the ability to adjust compute and memory resources according to workload changes, is a critical requirement for in-memory caching systems.

However, existing caching systems are constructed with and deployed on monolithic servers with coupled CPU and memory, which has two issues in dynamic resource adjustments. First, resource utilization is compromised since CPU and memory have to be added or reduced *together* as fix-sized virtual machines (VMs) on monolithic servers [22, 51]. While in practice, services may only want to add more memory or CPU cores to increase either cache capacity or request throughput. Besides, the speed of adjusting resources is too slow to cope with the workload bursts due to the time-consuming data migration [23, 38].

Disaggregated memory (DM) [2, 30, 41, 44, 63] is a promising approach to address these issues. It decouples the CPU and memory of monolithic servers into independent compute and memory pools and connects them with high-speed CPU-bypass interconnects, *e.g.*, remote direct memory access (RDMA) [31] and compute express link (CXL) [68]. CPUs and memory can thus be independently adjusted as application demands, improving resource efficiency. Moreover, the frequency of data migration can be greatly reduced since data are shared by all CPU cores in the compute pool and only need to be migrated to achieve better load balancing [40, 65]. However, two challenges have to be addressed to achieve a practical caching system on DM.

1) Bypassing remote CPUs hinders the execution of caching algorithms. Caching systems use various caching algorithms under different workloads [11, 58]. Caching algorithms monitor the hotness of cached objects and select eviction victims by maintaining the hotness information in caching data structures. Since data access changes object hotness, existing caching algorithms rely on the CPUs of caching servers, where all data accesses are executed, to monitor object hotness and maintain caching data structures [48]. However, in caching systems on DM, applications (clients) in the compute pool bypass CPUs in the memory pool when accessing objects. Evaluating object hotness becomes difficult due to the

[1] Work mainly done during the internship at Huawei Cloud.

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10...$15.00
https://doi.org/10.1145/3600006.3613144

lack of a centralized hotness monitor on data paths. Selecting eviction victims becomes inefficient since caching data structures have to be maintained with multiple high-latency remote memory accesses by clients, where data accesses are executed. Moreover, supporting various caching algorithms for different workloads [11, 58] is even more difficult on DM since caching algorithms evict objects with specified rules and tailored data structures [33, 73].

2) Adjusting resources affects hit rates of caching algorithms. Hit rates of caching algorithms relate to the data access patterns of workloads [74] and the cache size [59]. On DM, both attributes change on dynamical resource adjustments. The data access pattern changes with the number of concurrent clients (*i.e.*, compute resources), and the cache size changes with the allocated memory spaces (*i.e.*, memory resources). As a result, the best caching algorithm that maximizes hit rate changes dynamically with resource settings. Caching systems with fixed caching algorithms cannot adapt to these dynamic features of DM and can lead to inferior hit rates.

We address these challenges with Ditto[1], an elastic and adaptive caching system on DM. First, we propose a client-centric caching framework with *distributed hotness monitoring* and *sample-based eviction* to address the challenges of executing caching algorithms on DM. The distributed hotness monitoring uses one-sided RDMA verbs to record the access information from distributed clients in the compute pool, uses eviction priority to formally describe object hotness, and assesses objects' eviction priorities by applying priority calculation rules on the recorded access information. The sample-based eviction scheme selects eviction victims by sampling multiple objects and selecting the one with the lowest priority on the client side without maintaining remote data structures [58]. Since the key difference among caching algorithms is their definitions of eviction priorities, various caching algorithms can be integrated by defining tailored priority calculation rules with little coding effort. Second, we propose a distributed adaptive caching scheme to address the challenge of dynamic resource change. Ditto simultaneously executes multiple caching algorithms with the client-centric caching framework and uses regret minimization [26, 27, 85], an online machine learning algorithm, to perceive their performance and select the best one in the current resource setting.

We implement Ditto and evaluate its performance with both synthesized and real-world workloads [37, 67, 83]. Ditto is more elastic than Redis regarding resource efficiency and the speed of resource adjustments. On YCSB and real-world workloads, Ditto outperforms CliqueMap [66], the state-of-the-art key-value cache, by up to 9× and 3.6×, respectively. Moreover, Ditto can flexibly extend 12 widely-used caching algorithms with 12.5 lines of code (LOC) on average. The implementation of Ditto is open-source[2].

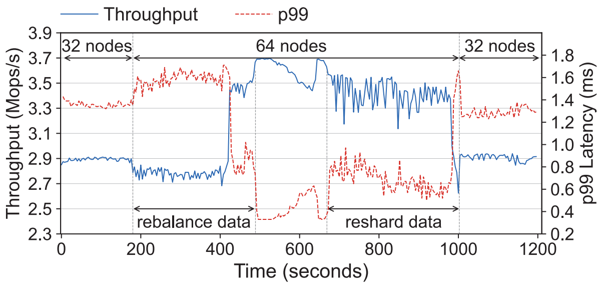


Figure 1. The performance of Redis when adjusting resources.

The contributions of this paper include the following:

- We identify the elasticity benefits and challenges of constructing caching systems on DM and propose Ditto, the first caching system on DM.
- We propose a client-centric caching framework where various caching algorithms can be integrated flexibly and executed efficiently on DM. A sample-friendly hash table and a frequency counter cache are designed to improve the efficiency of the framework on DM.
- We propose distributed adaptive caching to provide high hit rates by selecting the best caching algorithm according to the dynamic resource change and various data access patterns on DM. A lightweight eviction history and a lazy weight update scheme are designed to efficiently achieve adaptivity on DM.
- We implement Ditto and evaluate it with various workloads. Ditto outperforms the state-of-the-art approaches by up to 9× under YCSB synthetic workloads and up to 3.6× under real-world workloads.

2 Background and Motivation

2.1 Issues of Caching Systems on Monolithic Servers

There are two issues with existing caching systems on monolithic servers when they adjust resources.

1) Resource inefficiency. Resources of existing caching services on monolithic servers, *e.g.*, ElastiCache [22], are allocated with fix-sized virtual machines (VMs) with both CPU and memory, *e.g.*, 1 CPU with 2 GB DRAM, to facilitate resource management in monolithic-server-based datacenters. Resources are wasted when coupled CPU and memory are allocated, but only CPU or memory needs to be dynamically increased. Moreover, applications' demands on resources must be rounded up to fit in these fix-sized VMs, causing low resource utilization in the entire datacenter [8].

2) Slow resource adjustments. Existing in-memory caching systems shard data to multiple VMs to leverage more CPU and memory resources [22, 28, 43, 58]. Cached data have to be resharded and migrated when new VMs are added to the caching cluster. The migration cost [23] is unavoidable when

[1] Ditto is a Pokémon that can arbitrarily change its appearance.

[2] https://github.com/dmemsys/Ditto.git.

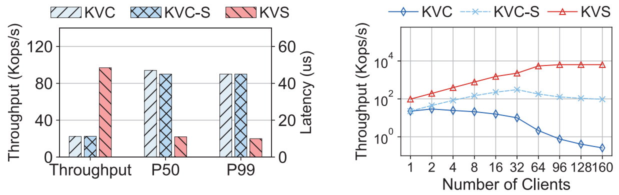
either CPU or memory needs to be adjusted due to the coupled allocation of CPU and memory on monolithic servers. The performance gain when increasing resources and the resource reclamation after shrinking resources is delayed for minutes due to the time-consuming data migration [39]. Moreover, the throughput drops and latency increases due to the consumption of additional CPU cycles and network bandwidths spent on moving data [38, 56].

Figure 1 shows the migration cost on Redis [58], the backend of many cloud caching services [22, 28], during resource adjustments under the read-only YCSB-C workload [17] with 10 million 256B key-value pairs. We first use 32 Redis nodes, each with 1 CPU core and 1 GB DRAM, then add 32 more nodes after 3 minutes of execution, and shrink back to 32 nodes after 3 minutes of stable execution with 64 nodes. We launch all 64 Redis nodes and idle 32 of them initially to rule out the cost of starting Redis nodes. We use 512 client threads to get the maximum throughput. When scaling to 64 nodes, Redis takes 5.3 minutes to migrate data. The throughput drops up to 7%, and the 99th percentile latency increases up to 21% in the process. When shrinking back to 32 nodes, the resource reclamation is delayed for 5.6 minutes due to data migration. Such migration cost is unavoidable even if using advanced migration techniques [23, 38] since CPU and memory are allocated in a coupled manner in VMs and objects are sharded to individual VMs.

2.2 Disaggregated Memory

Disaggregated memory (DM) is proposed to reduce the total cost of ownership (TCO) and improve the elasticity of applications on cloud datacenters [62, 63, 76]. It decouples compute and memory resources of monolithic servers into autonomous compute and memory pools. The compute pool contains compute nodes (CNs) with abundant CPU cores and a small amount of DRAM serving as run-time caches. The memory pool holds memory nodes (MNs) with adequate memory and a controller with weak compute power (*e.g.*, 1 - 2 CPU cores) to execute management tasks, *i.e.*, network connection and memory management. CNs and MNs are connected with CPU-bypass interconnects with high bandwidth and microsecond-scale latency, *e.g.*, RDMA and CXL [68], ensuring the performance requirements of memory accesses. CNs can allocate and free variable-sized memory blocks in the memory pool through the ALLOC and FREE interfaces provided by the controller. Without loss of generality, in this paper, we assume that CNs access MNs through one-sided RDMA verbs, *i.e.*, READ, WRITE, ATOMIC_CAS (compare and swap), and ATOMIC_FAA (fetch and add).

The decoupled compute and memory resources of DM addresses the resource efficiency and elasticity issues of existing caching systems. First, with DM, compute and memory resources can be allocated separately in a fine-grained manner [76]. Resources can be used more efficiently by assigning the exact amount of resources as per application demands.



(a) Single-client performance.

(b) Multi-client throughput.

Figure 2. The cost of maintaining caching data structures on DM.

Second, the frequency of data migration can be greatly reduced. Specifically, caching systems on DM do not need to migrate data when expanding or reducing memory since the cached data in the memory pool can be accessed by all CNs in the compute pool. Only in some special cases, *e.g.*, the network bandwidth of an MN becomes the performance bottleneck due to skewed workloads, data migration happens to achieve better load balancing. As a result, the migration cost can be eliminated for most cases, allowing resource adjustments to take effect agilely without performance losses.

3 Challenges

3.1 Executing Caching Algorithms on DM

Existing caching algorithms are designed for *server-centric* caching systems on monolithic servers where all data are accessed and evicted by the server-side CPUs in a centralized manner. Such a setting, however, no longer holds on DM because 1) caching systems on DM are *client-centric*, where clients directly access and evict the cached data in a CPU-bypass manner, and 2) the compute power in the memory pool of DM is too weak to execute caching algorithms on the data path. Two problems need to be addressed to execute caching algorithms on DM.

The first problem is how to evaluate the hotness of cached objects in the *client-centric* setting. Existing caching algorithms assess objects' hotness by monitoring and counting all data accesses [7, 12, 73]. The monitoring can be trivially achieved on server-centric caching systems since the CPUs of monolithic caching servers access all data. However, on DM, accesses to cached objects cannot be monitored either in the memory pool or on clients because 1) RDMA bypasses the CPUs in the memory pool, and 2) individual clients in the compute pool are not aware of global data accesses.

The second problem is how to efficiently select eviction victims on the client side. Caching algorithms maintain various caching data structures, *e.g.*, lists [73], heaps [7, 12], and stacks [32], to reflect the hotness of cached objects and select eviction victims based on these data structures. The data structures are maintained by the CPUs of caching servers on each data access since access changes object hotness. However, the maintenance of caching data structures has to be executed by clients in the compute pool since clients directly access objects with one-sided RDMA verbs. Maintaining these data structures thus becomes inefficient due to the multiple RTTs required on the critical path. Besides, locks are

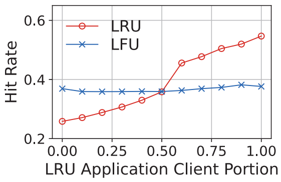


Figure 3. Hit rates under different numbers of clients under different applications.

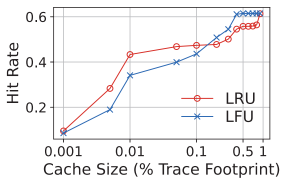
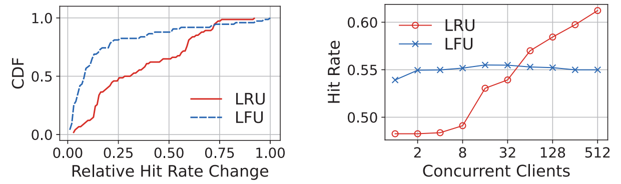


Figure 4. Hit rates of LRU and LFU on the same workload with different cache sizes.



(a) The CDF of relative hit rate changes on 74 workloads.

(b) Hit rates under different number of concurrent clients.

Figure 5. The effect of concurrent clients on hit rates.

required to ensure the correctness of caching data structures under concurrent accesses [48]. The throughput of caching systems will be severely bottlenecked by the microsecond-scale lock latency and the network contention caused by iteratively retying on lock failures [77].

To show the problem of maintaining caching data structures, we compare the performance of a linked-list-based LRU key-value cache (KVC), a key-value cache with sharded LRU lists (KVC-S), and a key-value store (KVS) on DM [65] under the read-only YCSB-C benchmark [17]. All approaches use a lock-free hash table to index cached objects. KVC maintains a lock-protected linked list to execute LRU. KVC-S shards the LRU list into 32 sub-lists to avoid lock contention and sleeps 5 us on lock failures to reduce the wasted RDMA requests on lock failures. Figure 2a shows the throughput and latency of the three approaches with a single client, ruling out lock contention. The throughput of KVC and KVC-S is only 23% of that of KVS, and the tail latency is more than 4.5× higher due to the additional RDMA operations on the critical path of data accesses. Figure 2b shows their throughput with growing numbers of client threads. The throughputs of KVC and KVC-S drop with more than 32 client threads because the RNIC of the MN is overwhelmed by the useless RDMA_CASes on lock-fail retries. The throughput of KVC-S drops more mildly due to the 5 us backoff on lock failures.

3.2 Dynamic Resource Changes Affect Hit Rate

Hit rates of caching algorithms closely relate to the data access patterns and the cache size [59]. However, both aspects are affected when dynamically adjusting compute and memory resources, making the best caching algorithm that maximizes the hit rate changes accordingly. Since DM enables resources to be adjusted fleetly and frequently, the effect of changing resource settings is amplified. Caching systems with fixed caching algorithms cannot adapt to these dynamic features of DM and can lead to inferior hit rates.

1) Changing compute resources affects hit rates. On caching systems on DM, applications execute multiple client threads on CPU cores in the compute pool to access cached data in the memory pool. The access pattern on cached objects is the mixture of access patterns of all applications. The change in compute resources, *i.e.*, the number of client threads of an application, alters the overall mixture of access patterns and affects the hit rate of individual caching algorithms in two ways.

First, the percentage of the data accesses of an application in the mixture changes with the number of client threads. The overall access pattern on the cached objects thus changes since applications have dissimilar access patterns [15]. Figure 3 shows the simulation result on a single machine with two applications under varying numbers of client threads. One application executes an LRU-friendly workload and the other executes an LFU-friendly one from the FIU block trace [37]. The hit rates of LRU and LFU are affected by the change of the compute resources in applications, where LFU exhibits a better hit rate when the LFU-friendly application has more compute resources and vice versa.

Second, the number of concurrent clients in an application changes the original access pattern of a workload due to concurrent executions. We simulate on 74 real-world workloads from Twitter [83] and FIU [37] with numbers of clients ranging from 1 to 512. Figure 5a shows the cumulative distribution function (CDF) of the relative hit rate change in these workloads. The relative hit rate change is calculated as $\frac{h_{max}-h_{min}}{h_{max}}$, where h_{max} and h_{min} are the highest and lowest hit rates of a workload under different numbers of clients. As we increase the number of client threads, 80% of workloads have 60% hit rate change in LRU and 21% in LFU. Meanwhile, the best caching algorithms on 36% of workload change with the varying number of concurrent clients. Figure 5b shows an example FIU trace where the hit rate of LFU performs better with a small number of concurrent clients but becomes inferior to LRU when the number of clients increases.

2) Changing memory resources affects hit rates. Changing memory resources leads to changing cache sizes of caching systems on DM. For individual workloads, the best caching algorithm that maximizes the hit rate changes with cache sizes [59], *e.g.*, one workload can be LRU-friendly with a small cache size but becomes LFU-friendly under bigger cache sizes. Our simulation finds that the best algorithm changes in 22 of the 74 real-world workloads when the cache size changes. Figure 4 shows an example FIU trace where LRU performs better with small caches and LFU performs better with larger cache sizes.

Consequently, it is necessary for caching systems on DM to dynamically select the best caching algorithm according to the changing resource settings. However, achieving adaptivity is difficult on DM due to its decentralized and distributed nature, as we will introduce in § 4.3.

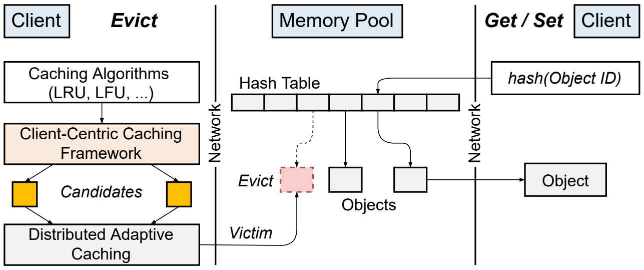


Figure 6. The overview of Ditto.

Table 1. The recorded access information.

Name	**Description**	Global?	Stateful?
size	Object size	✓	✗
insert_ts	Insert timestamp	✓	✗
last_ts	Last access timestamp	✓	✗
freq	The number of accesses	✓	✓
latency	Access latency	✗	✗
cost	Cost to fetch the object from the storage server	✗	✗

4 The Ditto Design

4.1 Overview

Figure 6 shows the overall architecture of Ditto. Ditto adopts a hash table to organize objects stored in the memory pool. The hash table stores pointers to the addresses of the cached objects. Following existing architectures of storage systems on DM [65, 66], applications execute on CNs and each application owns a local Ditto client as a subprocess. Each Ditto client has multiple threads on dedicated cores and applications communicate with Ditto clients with local shared memory to execute *Get* and *Set* operations. Under this architecture, applications can freely scale compute resources by adding or removing the number of threads and CPU cores assigned to Ditto. The adjustment on compute resources is independent against cached data because there is no need to increase or decrease the cache size in the memory pool when adding or reducing CPU cores.

Ditto clients execute *Get* and *Set* operations with one-sided RDMA verbs similar to RACE hashing [88], the state-of-the-art hashing index on DM. For *Get*s, a client searches the address of the cached object in the hash table and fetches the object from the address with two RDMA_READs. For *Sets*, a client searches the slot of the cached object in the hash table with an RDMA_READ, writes the new object to a free location with an RDMA_WRITE, and atomically modifies the pointer in the slot with an RDMA_CAS.

Ditto proposes a client-centric caching framework (§ 4.2) and a distributed adaptive caching scheme (§ 4.3) to achieve cache eviction on DM. The client-centric caching framework efficiently executes multiple caching algorithms on DM by selecting multiple eviction candidates of various caching algorithms. The distributed adaptive caching scheme uses machine learning to learn the characteristics of the current data access pattern and evicts the candidate selected by the caching algorithm that performs the best.

4.2 Client-Centric Caching Framework

The client-centric caching framework addresses the challenges of evaluating object hotness and selecting eviction candidates when executing caching algorithms on DM.

First, to assess the hotness of cached objects, Ditto records objects' access information and decides objects' hotness by defining and applying priority functions to the recorded access information. Specifically, Ditto associates each object with a small metadata recording its global access information, *e.g.*, access timestamps, frequency, etc. The metadata is updated collaboratively by clients with one-sided RDMA verbs after each *Get* and *Set*. On the client side, Ditto offers two interfaces to integrate caching algorithms, *i.e.*, priority functions (`double priority(Metadata)`) and metadata update rules (`void update(Metadata)`). A priority function maps the metadata of an object to a real value indicating its hotness. Since the key difference between caching algorithms is their definition of object hotness, various caching algorithms can be integrated by defining different priority functions with the `priority` interface. To support as many algorithms to be simply integrated with the `priority` interface as possible, we summarize the access information commonly used by existing caching algorithms [54] in Table 1 and record them in Ditto by default.

For advanced caching algorithms that require more access information, we allow algorithms to extend and define their own rules to update the metadata with the `update` interface. Listing 1 shows an example implementation of LRU-K [52]. LRU-K evicts objects with the smallest timestamp at its last K^{th} access. We split the *last_ts* field into K timestamps with lower precision and construct a ring buffer with the $freq$ counter. If the object is accessed more than K times, then its priority is its last K^{th} access timestamp, which is indexed by $(freq - K + 1) \bmod K$. Otherwise, we return the *insert_ts* of the object to achieve FIFO eviction in the access buffer [33]. We resort to storing the modified timestamp of LRU-K with cached objects if we need to simultaneously execute LRU-K with caching algorithms that rely on *last_ts*, *e.g.*, LRU.

Second, to efficiently select eviction candidates of various caching algorithms on DM, Ditto adopts sampling with client-side priority evaluation. The overhead of maintaining expensive caching data structures is then avoided. Specifically, on each eviction, Ditto randomly samples K objects in the cache and applies the defined priority functions to the access information of the sampled objects. The eviction victim is approximated as the object with the lowest priority among K sampled objects.

To efficiently execute the framework on DM, Ditto proposes a *sample-friendly hash table* and a *frequency-counter*

Listing 1. The pseudocode of LRU-K.

```
def update(Metadata m):
    m.freq += 1
    idx = m.freq % K
    m.last_ts[idx] = current_timestamp()

def priority(Metadata m):
    if m.freq < K:
        return m.insert_ts
    idx = (m.freq - K + 1) % K
    return m.last_ts[idx]
```

cache to reduce the overhead of sampling objects and recording access information on DM.

4.2.1 Sample-friendly hash table.

The sample-friendly hash table reduces the overhead of recording access information and sample objects on DM. Specifically, sampling objects on DM suffers from high access latency because multiple RDMA_READs are required to fetch the metadata of objects scattered in the memory pool. Moreover, updating access information affects the overall throughput because these additional RDMA operations consume the bounded message rate of RNICs in the memory pool.

The sample-friendly hash table co-designs the sampling process with the hash index to address these two problems. First, instead of storing all metadata together with objects, Ditto stores the most widely used metadata (*i.e.*, the default ones) together with the slots in the hash index but retains the metadata extensions required by advanced caching algorithms in objects. With the co-designed hash table, sampling can be conducted with only one RDMA_READ by directly fetching continuous slots with a random offset in the hash table. Second, Ditto reduces the number of RDMA operations on updating object metadata by organizing access information according to their update frequency. The well-organized access information enables multiple access information to be updated with a single RDMA_WRITE.

Hash table structure. Figure 7 shows the structure of the sample-friendly hash table. The hash table has multiple buckets with multiple slots. Each slot consists of two parts, *i.e.*, an atomic field and a metadata field. The atomic field is similar to the slot of Race Hashing [88], which is 8-byte in length and modified atomically with RDMA_CASes when objects are inserted, updated, or deleted. The atomic field contains a 6-byte *pointer* referring to the address of the object, a 1-byte *fp* (fingerprint) accelerating object searching, and a 1-byte *size* recording the size of the stored object. Similar to RACE hashing [88], we use a 1-byte size field and measure the sizes of objects in the granularity of 64B memory blocks. For large objects, Ditto stores the remaining data in a second memory block that links to the first one. The metadata field records the access information required by most caching algorithms, as summarized in Table 1. An additional *hash* field is recorded for the distributed adaptive caching scheme, which will be discussed in § 4.3.

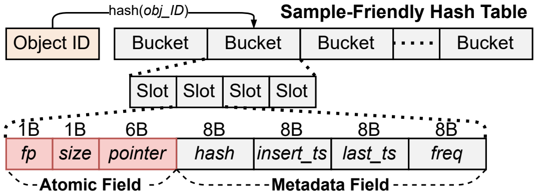


Figure 7. The sample-friendly hash table structure.

Access information organization. Ditto organizes the stored access information in two ways to reduce the number of RDMA operations on metadata updates. First, Ditto reduces the number of access information that has to be included in the metadata by distinguishing local and global information. Global information has to be maintained collaboratively by all clients and thus must be included in the metadata. Local information can be decided locally by distributed clients and hence does not need to be included. The *latency* and *cost* are local information because we assume that the latency and cost are approximately the same among clients and can be estimated based on the size of objects and the latency and cost of accessing other objects. Second, global information is further classified into stateless and stateful information. Stateless information is updated by overwriting its old value, while stateful information is updated based on its old value. For instance, the *insert_ts* and *last_ts* are stateless because the old timestamps are no longer useful. The *freq* is stateful because it is always updated to increase by 1. Ditto groups the stateless information together in the metadata so that they can be updated with a single RDMA_WRITE. The stateful information is updated with RDMA_FAAs.

4.2.2 Frequency-counter cache.

A client-side frequency-counter (FC) cache is proposed to further reduce the overhead of updating metadata. With the sample-friendly hash table, updating metadata still requires two RDMA operations, *i.e.*, an RDMA_WRITE to update the stateless information and an RDMA_FAA to update the stateful *freq*. These RDMA operations consume the message rate of the RNIC and thus limit the overall throughput of Ditto. Besides, executing RDMA_FAA on DM is expensive due to the contention in the internal locks of RNICs [35]. The FC cache aims to reduce the number of RDMA_FAA on metadata updates.

The FC cache stems from the idea of write-combining on modern processors [18]. In modern processors, several write instructions in a short time window are likely to target the same memory region, *e.g.*, a 64-byte cache line. The write combining scheme adopts a buffer to absorb writes to the same region in a short time window and convert them into a single memory write operation to save memory bandwidths.

Similar to write-combining, Ditto employs an FC cache as the write-combining buffer. The FC cache contains entries recording the object ID, the address of the slot in the hash table and the delta value of the counter. We track the insert time of each cache entry to ensure that the frequency

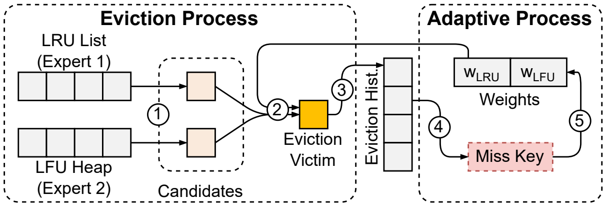


Figure 8. Adaptive caching on monolithic servers.

counters in the memory pool do not lag too much. Each time an object is accessed, its update to the frequency counter is buffered in the FC cache. The update to the remote frequency counter is deferred until a cache entry is evicted.

There are two situations when an entry will be evicted from the FC cache. First, if the space of the FC cache is full, an entry with the earliest insert timestamp will be evicted. Second, if the buffered delta value of an object is greater than a threshold t, the entry will be evicted. On entry eviction, the buffered counter value is added to the slot metadata with a single RDMA_FAA according to the recorded slot address, reducing the number of RDMA_FAA to up to $1/t$.

4.3 Distributed Adaptive Caching

Adaptive caching on monolithic servers is proposed to adapt to changing data access patterns in real-world workloads. Ditto proposes a distributed adaptive caching scheme to adapt to both changing workloads and dynamic resource settings on DM. The key problem is how to achieve adaptive caching in a distributed and client-centric manner on DM.

Recent approaches on monolithic servers formulate adaptive cache as a multi-armed bandit (MAB) problem [6, 47, 59, 74]. As shown in Figure 8, caching servers simultaneously execute multiple caching algorithms, named experts in MAB [6]. Each expert is associated with a weight, reflecting its performance in the current workload. The execution of the adaptive caching consists of an eviction and an adaptive process. During the eviction process, each expert proposes an eviction candidate according to their own caching data structures (①). Eviction victims are then decided opportunistically according to the weights of the experts (②), *i.e.*, candidates of experts with higher weights are more likely to be evicted. The metadata of the evicted object, *i.e.*, the object ID and the experts choosing it as a candidate, are inserted into a fix-sized FIFO queue named eviction history (③). During the adaptive process, existing approaches use *regret minimization* [26, 27, 85] to adjust expert weights. Specifically, finding a missed object ID in the eviction history is a *regret* because, intuitively, a more judicious eviction decision could have rectified the cache miss [74]. Hence, when the missed object ID is found in the eviction history (④), the weights of experts deciding to evict the object are decreased (⑤).

Two challenges have to be addressed to achieve adaptive caching on DM. First, maintaining the global FIFO eviction history is expensive due to the high overhead of accessing remote data structures on DM, as mentioned in § 3. Second, managing expert weights on distributed clients is costly since clients need to be synchronized to get the updated weights.

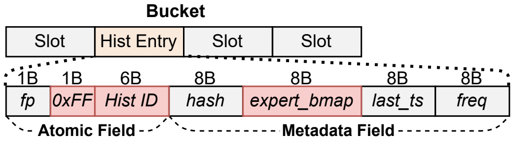


Figure 9. The structure of a lightweight history entry.

The distributed adaptive caching scheme addresses these DM-specific challenges. First, Ditto evaluates multiple priority functions with the client-centric caching framework to simultaneously execute multiple caching algorithms on DM. Second, to avoid maintaining an additional FIFO queue on DM, Ditto embeds eviction history entries into the hash table with a lightweight eviction history (§ 4.3.1). Finally, to efficiently update and utilize expert weights on the client side, Ditto proposes a lazy weight update scheme to avoid the expensive synchronization among clients (§ 4.3.2).

4.3.1 Lightweight eviction history. The eviction history on monolithic servers needs to maintain an additional FIFO queue and an additional hash index to organize and index history entries [59, 74]. The lightweight eviction history adopts two design choices to eliminate the overhead of maintaining these additional data structures on DM. First, it uses an *embedded history design* that reuses the slots of the sample-friendly hash table to store and index history entries. No additional space needs to be allocated and no additional hash index needs to be constructed for history entries. Second, the lightweight eviction history proposes a *logical FIFO queue with a lazy eviction scheme* to efficiently achieve FIFO replacement on history entries. No additional FIFO queue needs to be maintained to evict history entries.

Embedded history entries. Figure 9 shows the structure of an embedded history entry of the lightweight history. History entries are stored in the slots of the sample-friendly hash table with three differences. First, the *size* stores a special value (*0xFF*) to tag the slot as a history entry. We use *0xFF* instead of 0 since we use 0 to indicate empty slots. Second, the pointer field stores a 6-byte history ID instead of the address of the object. Finally, the history entry uses the *insert_ts* of the slot to store a bitmap indicating which experts have decided to evict the object (*expert_bmap*). Besides, each entry stores the hash value of the evicted object ID in the *hash* field to check if a missed object is contained in the eviction history. The hash value is written to the metadata when the object is inserted into the cache and will not be modified until its history entry is evicted from the FIFO eviction history.

The logical FIFO queue. The logical FIFO queue simulates FIFO eviction without actually maintaining a FIFO queue on DM. It is constructed with a global history counter

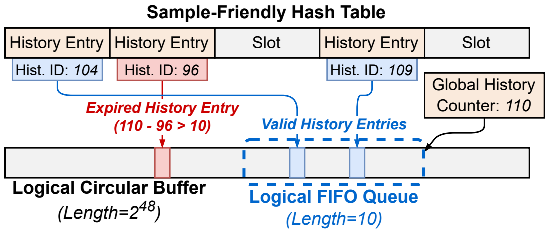


Figure 10. The logical FIFO queue structure.

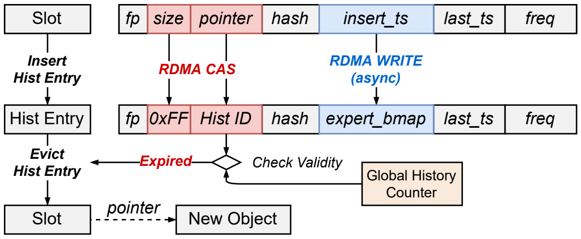


Figure 11. Inserting and evicting a history entry.

and the history IDs in history entries. The global history counter is a 6-byte circular counter that generates history IDs for new history entries. It is stored in an address in the memory pool known to all clients. The history IDs of history entries are acquired by atomically reading the global history counter and increasing it by one (*i.e.*, atomic fetch-and-add). As shown in Figure 10, the global history counter and history IDs of history entries can be viewed as locations in a logical circular buffer with 2^{48} entries. Combined with the size of the FIFO eviction history, the logical FIFO queue is then constructed, where the global history counter is the tail of the FIFO queue and the history IDs represent the location of history entries in the queue.

Figure 11 shows the operations of the lightweight history:

History insertion. A client inserts a history entry when it decides to evict a victim object from the cache. The client first acquires a history ID by performing an RDMA_FAA on the global history counter, which atomically returns the current value of the counter and increases it by one. Then the client issues an RDMA_CAS to atomically modify the *size* and the *pointer* in the slot of the victim object to be *0xFF* and the acquired history ID, respectively. The expert bitmap is then asynchronously written to the *insert_ts* field of the slot metadata with an RDMA_WRITE.

Lazy history eviction. Ditto adopts a lazy eviction scheme to achieve FIFO eviction on history entries, *i.e.*, expired history entries are kept in the history for a while before their evictions. To prevent clients from accessing expired history entries, Ditto proposes a client-side expiration checking mechanism. Suppose the global history counter is v_1, the history ID is v_2, and the size of the FIFO history is l. If $v_1 > v_2$, the history entry is invalid when $v_1 - v_2 > l$. Otherwise, the history entry is invalid if $v_1 + 2^{48} - v_2 > l$, considering the wrap-up of the 48-bit global history counter. The actual evictions happen when inserting new objects into the cache. As shown in Figure 11, when inserting new objects, the expired slots are considered empty slots and are overwritten to be ordinary slots, which transparently evicts the history entry.

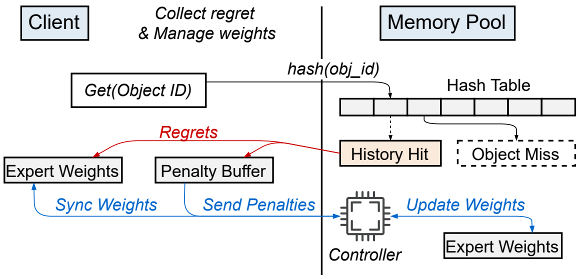


Figure 12. The lazy weight update scheme.

Regret collection. A regret is defined as a client finding an object to be missed in the cache but contained in the eviction history. The embedded history entry makes collecting regrets the same process as searching objects in the cache. When a client searches for an object, it calculates the hash value of the object ID, locates a bucket based on the hash value, and iteratively matches the slots in the bucket to see if the pointed object has the same object ID as the target. During the process, clients also match the hash value of the encountered history entries in the bucket. Regrets can then be collected if the object has not been found but a history entry has a matching hash value.

4.3.2 Lazy expert weight update.

Ditto formulates the problem of cache replacement as MAB and uses regret minimization to dynamically adjust the weights of experts. When a regret is found, *i.e.*, a missed object hits in the eviction history, the weights of the experts that evicted the object should be penalized. Suppose expert E_i made a bad eviction decision and the decision is the t-th entry in the eviction history. The weight of the expert is then updated to be $w_{E_i} = w_{E_i} \cdot e^{\lambda * d^t}$, where λ is the learning rate and d^t is the penalty. The penalty $e^{\lambda * d^t}$ is related to the position of the entry in the FIFO history because an older regret should be penalized less, where d is a fixed discount rate[3]. The challenge of updating weights on DM is that regrets are no longer collected and expert weights are no longer used in a centralized manner by monolithic caching servers. Updating and using expert weights from distributed clients incurs nonnegligible overhead due to the high synchronization overhead on DM [72].

The idea of the lazy weight update scheme is to let clients batch the regrets locally and offload the weight update lazily to the controllers of MNs. In this way, the frequency of updating weights is reduced and the overhead of synchronization is avoided. Meanwhile, the weak controller of memory nodes will not become a bottleneck due to the infrequent update.

Figure 12 shows the process of the lazy expert weight update scheme. Each client maintains expert weights locally to make eviction decisions. When a client discovers a regret,

[3] Similar to [74], the discount rate is $0.005^{1/N}$, where N is the cache size.

it applies the penalty to the local expert weights according to the history bitmap in the history entry. The penalties are recorded in a penalty buffer. When the number of buffered penalties exceeds a threshold, the client sends all the penalties to the controller of the memory node holding the expert weights with an RDMA-based RPC request. On receiving clients' penalties, the controller of the MN first applies the penalties to the global expert weights and then replies the updated global weights to clients.

To reduce the bandwidth consumption of transferring the penalties over the network, Ditto compresses the penalties using the attribute of exponential functions. Specifically, the sum of the penalties is stored in the penalty buffer and transferred to the MN instead of a list of individual penalties.

With the lazy weight update scheme, clients' eviction decisions are made on local weights, which are not always synchronized with global weights. However, such asynchrony does not affect the adaptivity of Ditto, as shown in our experiments.

4.4 Discussions

Metadata extensions. As mentioned in § 4.2.1, Ditto stores extended metadata together with cached objects for advanced caching algorithms. In this situation, the extended metadata is stored as a metadata header ahead of each object. The `update` and `priority` functions take all metadata, *i.e.*, the default ones in the hash table and the extended ones in the metadata header, as input and call user-defined metadata update and priority calculation rules to deal with the extended metadata. After executing *Get* and *Set* operations, an additional RDMA_WRITE is required to update the metadata stored with objects asynchronously. Finally, on cache eviction, additional RDMA_READs are required to fetch the metadata header to calculate eviction priorities.

Metadata overhead. In Ditto, metadata consists of history entries, the index slots for cached objects and global expert weights. First, each history entry contains 40 bytes, as shown in Figure 9. The total number of history entries is set as the maximum number of cached objects according to existing approaches [59, 74]. Second, for each cached object, the index slot uses 40 bytes, *i.e.*, 8 bytes for the atomic field and 32 bytes for access information, as shown in Figure 7. Finally, for each expert, a 4-byte float variable is required as its global expert weight. Summing up all of these, the metadata overhead of Ditto is $80 \cdot C + 4 \cdot N$ bytes, where C is the maximum number of cached objects and N is the number of experts in the distributed adaptive caching scheme.

Security and fairness issues. Since Ditto clients and applications cooperate closely on the same CNs, it is possible that some malicious users can manipulate Ditto clients to make them disproportionately advantaged against other users' applications. We can enforce security techniques, *e.g.*, control flow integrity (CFI) [1], on standalone Ditto clients to prevent Ditto clients from being manipulated. We can also integrate the expected delaying technique [55] in Ditto clients to ensure that applications fairly share the cache.

Table 2. Real-world workloads used in the evaluation.

Workload	Workload Type	# Requests
IBM	Object Store	10 - 40 million
CloudPhysics	Block IO	50 million
Twitter-Transient	Transient key-value cache	10 million
Twitter-Storage	Storage key-value cache	10 million
Twitter-Compute	Compute key-value cache	10 million
webmail	Block IO	7.8 million

5 Evaluation

The evaluation of Ditto answers the following questions:

- **Q1:** How elastic is Ditto compared with caching systems on monolithic servers?
- **Q2:** How efficient is Ditto in executing caching algorithms on DM?
- **Q3:** How adaptive is Ditto to real-world workloads and the changing resources on DM?
- **Q4:** How flexible is Ditto in integrating various caching algorithms on DM?
- **Q5:** How does each design point contribute to Ditto?

5.1 Experimental Setup

Testbed. We evaluate Ditto with 9 physical machines (8 CNs and 1 MN) on the Clemson cluster of CloudLab [20]. Each machine has two 36-core Intel Xeon CPUs, 256 GB DRAM, and a 100Gbps Mellanox ConnectX-6 NIC. All machines are connected to a 100Gbps Ethernet switch. In all our experiments, we use a single physical machine and use one CPU core to simulate the memory pool of DM with weak compute power [65, 72]. Ditto is compatible with memory pools with multiple MNs as long as the memory pool offers the required interfaces presented in §2.2. Besides, we use up to 32 cores on CNs, with each executing a client thread because they are on the same NUMA node with the RNIC.

Workloads. We evaluate Ditto with both YCSB synthetic workloads [17] and real-world key-value traces [37, 67, 83]. For YCSB synthetic workloads, we use 4 core workloads: A (50% GET, 50% UPDATE), B (95% GET, 5% UPDATE), C (100% GET), and D (95% GET, 5% INSERT). For all four workloads, we pre-generate 10 million keys with 256-byte key-value pairs, load these generated keys by sharding them to all clients, and execute the corresponding workloads. The requests are generated with Zipfan distribution with $\theta = 0.99$. For real-world key-value traces, we use workloads from IBM [24], CloudPhysics [75], Twitter [83], and FIU [37], as shown in Table 2. The *IBM* trace is collected from IBM Cloud Object Storage [24]. We ignore traces with less than 10 million requests since they have too few unique objects and use all 23 traces in our experiments. The *CloudPhysics* dataset includes block I/O traces on VMs with different CPU/DRAM configurations [75]. We use the first 10 traces with more than 50 million requests to evaluate Ditto. For the Twitter traces, we randomly select three traces, *i.e.*, *Twitter-Compute*, *Twitter-Storage*, and *Twitter-Transient*, from a compute cluster, a storage cluster, and a transient caching cluster, respectively. The *webmail* trace is a 14-day storage I/O trace collected from web-based email servers. We use *webmail* as a representative FIU trace similar to existing approaches [59]. In our experiments, we randomly select traces to accelerate our evaluation to show the performance of Ditto in different use cases, *i.e.*, block IO, KV cache on different clusters, and object store. We truncate traces to allow concurrent trace loading from 32 independent clients on a single CN.

Implementations. We implement Ditto with 20k LOCs. We use LRU and LFU, the two most widely used caching algorithms, as two experts in the distributed adaptive caching scheme. These two caching algorithms are chosen as adaptive experts since existing adaptive caching schemes have found that using a recency-based and a frequency-based caching algorithm can adapt to most workloads [59, 74]. For memory management, we use a two-level memory management scheme [65] so that clients can dynamically allocate memory spaces in the MN. We pre-register all memory on the MN to its RNIC to eliminate the overhead of memory registration on the critical path of memory allocation.

Parameters. The parameters of Ditto include the number of samples, the size of lightweight eviction history, the threshold and size of the FC Cache, and the learning rate and the number of batched weight updates of distributed adaptive caching. Specifically, the number of samples affects the precision of approximating caching algorithms with sampling. We sample 5 objects on cache eviction according to the default value of Redis [58]. The size of the lightweight eviction history exhibits a tradeoff between the speed of adaptation and the metadata overhead. Setting the history size larger makes adaptation faster since more penalties can be collected during execution. In return, a larger history size requires more space to store history entries. We set the history size as the cache size (calculated in the number of objects) according to LeCaR [74]. The threshold of FC Cache can affect the precision of LFU. We set the FC cache threshold to 10 and set the FC cache size to 10MB according to our grid search. The superior hit rates in our experiments show that using 10 as the FC threshold does not affect hit rates much. Finally, we configure the learning rate of Ditto to be 0.1 and update global weights every 100 local weight updates according to our grid search.

Baselines. We compare Ditto with Redis [58], CliqueMap [66], and Shard-LRU. First, we use Redis, one of the most widely adopted in-memory caching systems that support dynamic resource scaling [22, 58], to show the elasticity of Ditto. Second, we use CliqueMap, the state-of-the-art RDMA-based KV cache from Google, to show the efficiency and adaptivity of Ditto. CliqueMap initiates RDMA_READs on the client side to directly *Get* cached objects, and relies on server-side CPUs to execute *Set* operations. Since *Get*s involves only one-sided RDMA_READs, no access information can be recorded. Clients of CliqueMap record access information locally and send the information to servers periodically to enable servers to execute caching algorithms. We implemented an LRU (CM-LRU) and LFU (CM-LFU) version of CliqueMap according to its paper due to no open-source implementations. We disable the replication and fault-tolerance of CliqueMap to focus on comparing the execution of caching algorithms. Finally, we use Shard-LRU, a straightforward implementation of a caching system on DM, to show the effectiveness of the client-centric caching framework of Ditto. Clients of Shard-LRU maintain lock-protected LRU lists in the memory pool with one-sided RDMA verbs. We shard objects into 32 LRU lists according to their hash values and force clients to sleep 5 us on lock failures to mitigate lock and network contention. By default, we use one CPU core on MNs to simulate the poor compute power in the memory pool. Each CPU core on CNs exclusively runs a client thread.

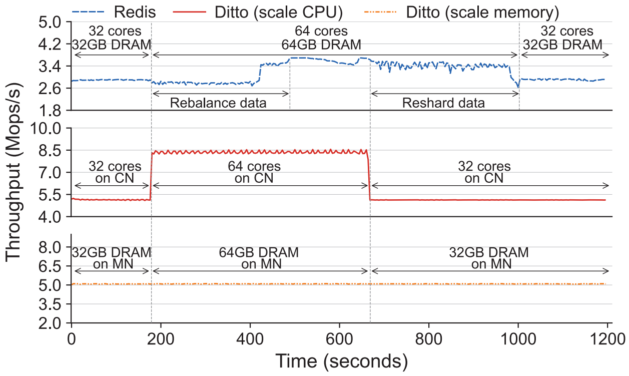
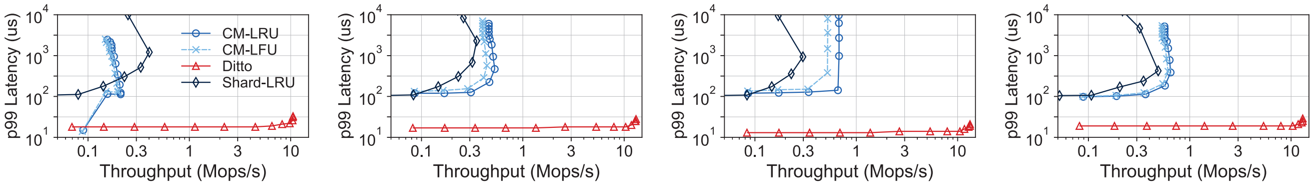


Figure 13. The throughput of Ditto when dynamically adjusting compute and memory resources.

5.2 Q1: Elasticity

To show the elasticity of Ditto, we run the same experiment as in § 2 and force Ditto to use the same amount of CPU or memory resources as Redis on the YCSB-C workload.

Compared with Redis, the elasticity of Ditto is improved in both resource utilization and speed of resource adjustments. First, due to the decoupled CPU and memory on DM, Ditto can adjust CPU cores and memory spaces separately in a fine-grained manner. Resources can be allocated precisely according to the dynamic demands of applications. Second, Ditto does not require data migration when adjusting resources, making the performance gain and resource reclamation more agile than Redis. The throughput of Ditto improves immediately from 5 Mops to 8.5 Mops with 32 more CPU cores added and resumes immediately back to 5 Mops as we shrink the number of CPU cores back to 32. The throughput doesn't scale linearly as we add CPU cores due to the extra overhead of coroutine scheduling on CNs. The



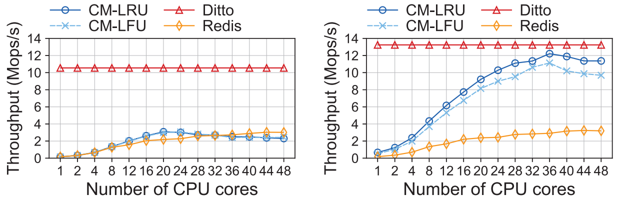
(a) YCSB A

(b) YCSB B

(c) YCSB C

(d) YCSB D

Figure 14. The throughput and tail latency of caching systems on DM.



(a) YCSB A (write-intensive)

(b) YCSB C (read-only)

Figure 15. The throughput of CliqueMap, Redis, and Ditto with more CPU cores on MN.

median latency stabilizes at 12 us and the 99th percentile latency fluctuates slightly around 14 to 21 us. As for adjusting memory spaces, the throughput stabilizes on 5 Mops and the tail latency stays on 14 us. Besides, the throughput of Ditto is more than 2 times higher than that of Redis during the entire experiment. This is because Ditto allows CPU cores to equally access all data, avoiding a single core becoming the performance bottleneck. However, Redis shards data to VMs, which makes the CPU core of some VMs bottleneck the throughput of the entire caching cluster on the skewed YCSB workloads.

Besides, Ditto does not require more client-side computation than Redis. In the experiment, clients of Ditto consume 32 CPU cores on the CN. In contrast, clients of Redis consume on average 36.3 CPU cores out of 128 assigned cores on two CNs. This is because the Redis client library spends CPU cycles to encapsulate and decapsulate data according to the Redis communication protocol and network protocols. Moreover, Ditto saves compute power regarding the overall CPU utilization since Redis servers consume an additional 32 cores on the MN.

5.3 Q2: Efficiency

To show that Ditto can efficiently execute caching algorithms on DM, we evaluate the throughput and tail latency of Shard-LRU, CliqueMap, and Ditto in the case of no cache misses on YCSB benchmarks. We vary the number of clients from 1 to 256, with each CN holding up to 32 clients.

As shown in Figure 14, Shard-LRU is bottlenecked by its remote lock contention even if the sharded LRU list and the 5 us back-off scheme mitigate the lock and network contention. The throughput of CliqueMap is limited by the weak compute power on MNs. For write-intensive workloads (YCSB A), the CPU of the MN is overwhelmed by frequent *Set*s. For read-intensive workloads (YCSB B, C, and D), the CPU of the MN is busy with merging the object access information received from clients. The overall performance is affected by the periodic synchronization of access information and the amplified network bandwidth when sending the access information from clients to the MN.

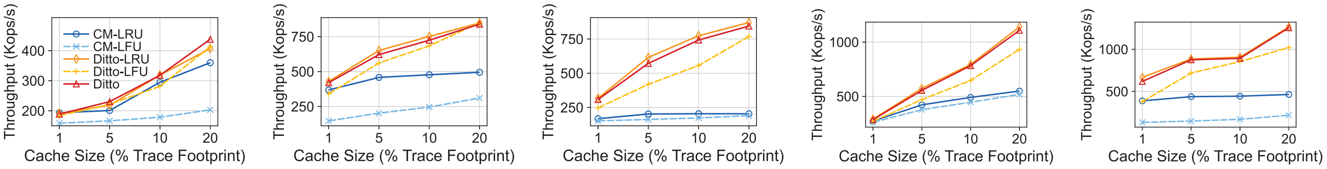
For all workloads, Ditto is bottlenecked by the message rate of the RNIC on the MN. It achieves 10.5, 13.1, 13.2, and 13.0 Mops respectively on YCSB A, B, C, and D workloads, which is up to 9× higher than Shard-LRU and CliqueMap. Compared with Shard-LRU, Ditto records the access information and selects eviction victims in a lock-free manner, eliminating the expensive lock overhead on DM. Compared with CliqueMap, Ditto accesses data and maintains access information with one-sided RDMA verbs, preventing the weak compute power on the MN from becoming the throughput bottleneck on both write-intensive and read-intensive workloads. However, Ditto performs worse than CliqueMap under the write-intensive YCSB-A workload with a single client, *i.e.*, the first point in Figure 14a. This is because the *Set*s of CliqueMap use only a single RTT, while Ditto needs three RTTs to search the remote hash table, read the object, and modify the pointer in the hash table.

Figure 15 shows the performance of CliqueMap, Redis and Ditto under YCSB-A and YCSB-C workloads with increasing numbers of MN-side CPU cores under 256 clients. We shard the LRU list (and the LFU heap) of CliqueMap into 128 shards to avoid server-side lock contention. The throughput of Ditto stays the same since Ditto does not rely on compute power on MNs. With the same compute resource in the compute pool, CliqueMap consumes more than 20 additional cores to get comparable performance with Ditto on YCSB-C. Ditto achieves 3.3× higher throughput than CliqueMap on the write-intensive YCSB-A workload since CliqueMap relies only on the server-side compute power to execute *Set* operations and maintain caching data structures. The throughputs of Redis on both workloads are bottlenecked by the CPU core of the hottest data shard due to the skewed YCSB workloads. Redis performs slightly better than CliqueMap on YCSB-A workload with more CPU cores since its sample-based eviction eliminates the overhead of maintaining caching data structures locally.

5.4 Q3: Adaptivity

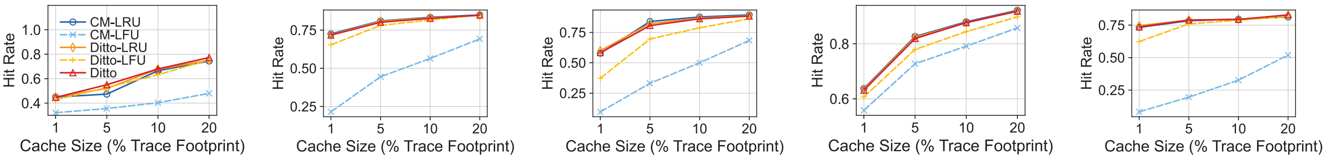
5.4.1 Adapt to real-world workloads.

To show the adap-



(a) Webmail **(b)** Twitter-Transient **(c)** Twitter-Storage **(d)** Twitter-Compute **(e)** IBM

Figure 16. Penalized throughputs under different real-world workloads.



(a) Webmail **(b)** Twitter-Transient **(c)** Twitter-Storage **(d)** Twitter-Compute **(e)** IBM

Figure 17. Hit rates under different real-world workloads.

tivity of Ditto on real-world workloads with different affinities of caching algorithms, we evaluate the throughput and the hit rate of real-world workloads with different cache sizes. For all traces, we use 256-byte object sizes and set cache sizes relative to the size of each workload's footprint, *i.e.*, all unique data items accessed, similar to [59]. For each workload, we use 64 clients to first execute 10 seconds to warm up the cache and then let all clients iteratively run the workload for 20 seconds to calculate the hit rate and the throughput. We use a penalized throughput to simulate real-world situations where caching systems cooperate with a distributed storage system. For each *Get* miss, we force clients to sleep for 500 us before inserting the missed object into the cache with *Set*. The penalty simulates the overhead of fetching data from distributed storage services and 500 us is selected according to the latency of the state-of-the-art distributed storage systems [46, 53, 81].

We compare Ditto with four baseline approaches. We use CM-LRU and CM-LFU to show the performance of precise LRU and LFU implementation with CliqueMap on DM. We introduce Ditto-LRU and Ditto-LFU to show the performance of Ditto with only a single caching algorithm.

Since Ditto is an adaptive caching framework that can execute various caching algorithms and dynamically adapt to the best one based on workloads and resource settings, the performance of Ditto largely depends on the candidate caching algorithms configured by users. We configure Ditto to execute LRU and LFU as examples to show its adaptivity. Under workloads that are friendly to either LRU or LFU, the performance of Ditto should be bounded by Ditto-LRU and Ditto-LFU and approach to the better one since it adaptively selects the better one among the two algorithms.

Figures 16 and 17 show the penalized throughput and the hit rates under five real-world key-value traces. In all five workloads, the hit rate and penalized throughput of Ditto can effectively approach the better one of Ditto-LRU and Ditto-LFU. Meanwhile, Ditto outperforms CliqueMap in all workloads due to higher hit rates and the higher throughput upper-bound. Particularly, the throughput of CliqueMap is bounded by the compute power on the MN under the Twitter workloads, where the hit rates are high. One exception is the throughput of CM-LRU in Figure 16a, which has comparable throughput with Ditto. This is because all approaches are bounded by the hit rate on the *webmail* workload and CM-LRU has a slightly lower hit rate compared with Ditto. For most of the workloads, the throughput of Ditto is lower than that of Ditto-LRU when their hit rates are the same due to the additional overhead of adaptive caching, *i.e.*, accessing and increasing the global history counter. However, the overhead is less than 5%, which is acceptable compared with the up to 63% performance gain of using an inferior caching algorithm, since users do not know in advance which caching algorithm performs better.

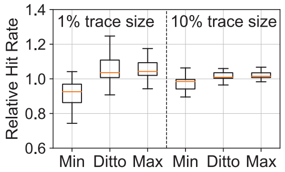


Figure 18. The relative hit rate of Ditto, Ditto-LRU, and Ditto-LFU on 33 workloads.

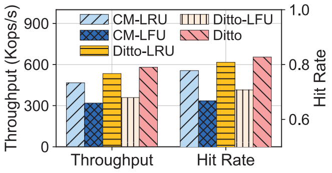


Figure 19. The penalized throughput and hit rate under a changing workload.

Figure 18 shows the box plot of relative hit rates of Ditto, max(Ditto-LRU, Ditto-LFU), and min(Ditto-LRU, Ditto-LFU) normalized over random eviction on 33 *IBM* and *CloudPhysics* workloads. The hit rate of Ditto significantly exceeds min(Ditto-LRU, Ditto-LRU) and approaches the box of max(Ditto-LRU, Ditto-LFU), showing the adaptivity of Ditto.

Under changing workloads that iteratively switch between LRU- and LFU-friendly, Ditto should outperform both Ditto-LRU and Ditto-LFU. We show the performance of the four approaches on a synthetic changing workload used in [74].

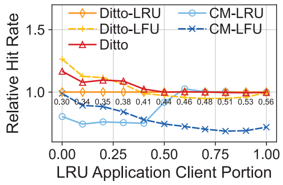


Figure 20. The relative hit rates under different proportions of clients assigned to LRU and LFU applications.

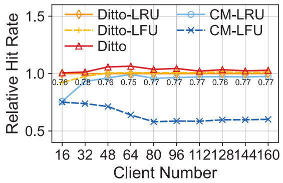


Figure 21. The relative hit rates of Ditto and CliqueMap when dynamically adding the number of concurrent clients.

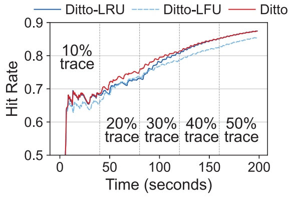


Figure 22. The hit rate under dynamic cache sizes.

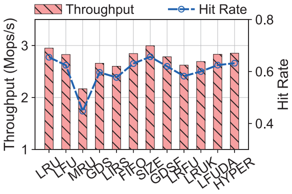


Figure 23. The throughput and hit rates of 12 algorithms.

The workload is synthesized to have four phases that periodically switch back and forth from being favorable to LRU to being favorable to LFU. As shown in Figure 19, Ditto outperforms all baselines on both penalized throughput and hit rate because only Ditto can adapt to workload changes.

5.4.2 Adapt to dynamic resource adjustments.

To show the adaptivity of Ditto on DM, we evaluate its hit rates with dynamically changing compute and memory resources on the same workload as Figures 3, 4, and 5b, *i.e.*, *webmail*.

Adapt to changing compute resources. Figure 20 shows the relative hit rates normalized to Ditto-LRU under different proportions of clients allocated to two applications with LRU and LFU access patterns. The hit rate of Ditto-LFU is higher when the LRU portion is less than 0.4, while Ditto-LRU performs better when the LRU portion grows higher. The hit rate of Ditto is higher than that of Ditto-LRU with a low LRU portion and becomes close to Ditto-LRU with a high LRU portion, indicating the adaptivity of Ditto. Besides, Ditto can adapt to the change of access pattern when multiple clients concurrently execute the same workload. Figure 21 shows the relative hit rates of Ditto and CliqueMap normalized to Ditto-LRU under dynamically increasing numbers of concurrent clients[4]. The hit rate of Ditto stays above the hit rates of both Ditto-LRU and Ditto-LFU because there are access pattern changes in the real-world *webmail* workload, and only Ditto can adapt to these changes.

Adapt to changing memory sizes. Figure 22 shows the hit rate of Ditto when we dynamically increase the memory space. The hit rate of Ditto approaches Ditto-LRU for most cases, outperforming Ditto-LFU. When the cache size is 20% and 30% footprint size, the hit rate of Ditto-LFU exceeds Ditto-LRU. Ditto performs better than both approaches because it can adaptively adjust its algorithm according to the affinity of caching algorithms on different cache sizes.

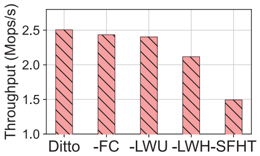


Figure 24. Contributions of different techniques on the *webmail* workload.

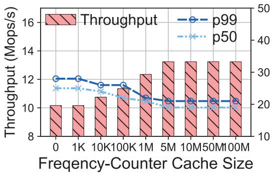


Figure 25. The YCSB-C performance of Ditto with different FC Cache sizes.

5.5 Q4: Flexibility

To show that Ditto can flexibly integrate various caching algorithms, we integrate 12 commonly used caching algorithms into Ditto and evaluate their throughput, hit rate, and coding effort. Since evaluating the feasibility of executing different caching algorithms is independent of workloads, we only show the throughput and hit rates on the *webmail* workload in Figure 23. Among all the algorithms, SIZE exhibits the best throughput and hit rate, while MRU exhibits the worst. All these algorithms can be easily implemented in Ditto with less than 23 lines of code, as shown in Table 3.

5.6 Q5: Contribution of Each Technique

We show the contribution of techniques proposed in the paper by gradually disabling each technique of Ditto. Due to the space limit, we show the performance of different techniques without miss penalties on the *webmail* workload in Figure 24. Ditto performs similarly on other workloads and more results can be found in our open-source repository. The sample-friendly hash table (SFHT) improves the overall throughput by 42% since it reduces the number of RDMA operations on data paths when updating the access information and sampling objects. The lightweight history scheme (LWH) improves the throughput by 13% due to the reduced number of RTTs when collecting regrets and maintaining eviction history. Finally, the lazy weight update scheme (LWU) and the frequency-counter cache (FC) contribute to 4% of the overall throughput because the reduced number of RDMA requests saves the message rate of the RNICs on MNs.

Figure 25 shows the performance of Ditto under the YCSB-C benchmark with 256 clients and different FC cache sizes. We limit FC cache size in MB since the size of each cache entry varies with the size of its recorded object ID. We only show the result under YCSB-C due to the space limit. Ditto performs similarly on other workloads and more results can be found in our open-source repository. The throughput increases from 10 Mops to 13.2 Mops with increased sizes of the FC cache since more RDMA_FAAs can be cached locally to save the message rate of RNICs. The tail latency drops from 28 us to 21 us due to the reduced number of

[4] The absolute hit rates in Figures 18, 20, and 21 can be found in our open-source repository.

Table 3. LOCs and used access information of different caching algorithms on Ditto. ts_I and ts_L refer to the insert timestamp and the last access timestamp, respectively. S refers to the size of the object, F refers to the access frequency of the object, and M refers to the use of additional metadata. Details on the additional metadata M can be found in our open-source repository.

Algs.	LRU	LFU	MRU	GDS	LIRS	FIFO	SIZE	GDSF	LRFU	LRUK	LFUDA	HYPERBOLIC
LOC	9	9	9	14	12	9	9	14	17	23	14	11
Info.	ts_L	F	ts_L	S	F, ts_L, M	ts_I	S	F, S	ts_L, M	M	F, M	ts_L, F, S

RDMA operations and less contended network. Also, the performance gain of the FC cache becomes insignificant when the size of the FC cache exceeds 5 MB, indicating that the FC cache can improve overall performance with small additional memory consumption on clients.

6 Related Work

Disaggregated Memory. Existing work on disaggregated memory can be classified into approaches that realize efficient memory disaggregation and approaches that design better applications. The realization approaches use software-based [3, 5, 19, 29, 51, 60, 63, 76], hardware-based [44, 68, 70], and hybrid [30, 41, 64, 78] techniques to efficiently achieve general-purpose disaggregated memory. Ditto is orthogonal to these approaches since Ditto only assumes the underlying DM to be capable of executing `READ`, `WRITE`, `CAS`, and `FAA`. Approaches that port applications on DM design important cloud applications, *e.g.*, key-value stores [40, 65, 72], transactional storage systems [87], and data structures [4, 42, 45, 77, 88], to achieve better resource efficiency. The work most related to Ditto are memory-disaggregated key-value stores, *i.e.*, Clover [72], Dinomo [40], and FUSEE [65]. However, all of them focus only on improving the performance for persistent and reliable data storage on DM, while Ditto is the first caching system that can efficiently execute various caching algorithms and adaptively select the best one on DM.

In-Memory Caching Systems. Many approaches aim at improving the performance of Memcached [48] and Redis [58], the two most popular in-memory caching systems. Some [13, 14, 61] optimize the hit rate under objects of varying sizes. Others [25, 43, 50, 57, 84] improve memory efficiency and overall throughput. The work closest to Ditto is CliqueMap [66], an RDMA-based caching system. It uses one-sided RDMA_READ for *Get* operations and RPC for *Set* operations, improving the throughput due to the higher bandwidth and CPU-bypass nature of one-sided RDMA_READ. However, all these approaches are designed and optimized for monolithic servers, which inevitably inherit the elasticity issues of monolithic servers. Ditto exhibits better elasticity by leveraging the hardware benefits of DM.

RDMA-Based KV stores. There are two types of RDMA-based KV stores, *i.e.*, server-centric and hybrid ones. The former uses RDMA to construct fast RPC primitives and rely on server CPUs to access data [19, 34, 36, 43]. The latter uses one-sided RDMA verbs to execute *Get* operations and relies on server CPUs to execute *Set* operations [49, 79, 80]. Compared with these approaches, Ditto achieves efficient in-memory caching without relying on server-side CPUs. Besides, the design of Ditto is not limited to RDMA. Other interconnects are also compatible.

Caching Algorithms. Caching algorithms distinguish the hotness of objects using recency [73, 86], frequency [21] and other access information [11], or combining various information together [7, 10–12] to get higher hit rates. Recently, there are many machine-learning-based adaptive caching algorithms [6, 47, 59, 74]. Among them, CACHEUS [59] is the most related. It uses regret minimization to adaptively select a better caching algorithm. However, all these caching algorithms are designed for server-centric caching systems to optimize specific workloads. Ditto, on the one hand, is designed for caching systems on DM where clients directly access data without involving CPUs in the memory pool. On the other hand, Ditto is an adaptive caching framework where multiple caching algorithms can be integrated and adaptively selected according to workload and resource change.

7 Conclusion

We propose Ditto, the first caching system on the disaggregated memory architecture, to achieve better elasticity. Ditto addresses the challenges of constructing a caching system on DM, *i.e.*, executing server-centric caching algorithms and dealing with inferior hit rates caused by dynamically changing resources and data access patterns. A client-centric caching framework is proposed to efficiently execute caching algorithms on DM. Various caching algorithms can be integrated with small coding efforts. A distributed adaptive caching scheme is proposed to adapt to the resource and workload changes. Experimental results show that Ditto effectively adapts to the resource and workload change on DM and outperforms the state-of-the-art caching system on monolithic servers by up to 9× on YCSB synthetic workloads and 3.6× on real-world key-value traces.

Acknowledgments

We sincerely thank our shepherd Marcos K. Aguilera and the anonymous reviewers for their constructive comments and suggestions. This work is supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund), the National Natural Science Foundation of China (Project No. 62202511), the Natural Science Foundation of Shanghai (Project No. 22ZR1407900), and Huawei Cloud. Pengfei Zuo is the corresponding author (pfzuo.cs@gmail.com).

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, 2009.

[2] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. Memory Disaggregation: Why Now and What are the Challenges. *ACM SIGOPS Operating Systems Review*, 57(1):38–46, 2023.

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. uBFT: Microsecond-Scale BFT using Disaggregated Memory. In *the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2023, March 25-29, 2023*, pages 862–877. ACM, 2023.

[4] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Workshop on Hot Topics in Operating Systems, HotOS 2019, May 13-15, 2019*, pages 120–126. ACM, 2019.

[5] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *the 15th EuroSys Conference, EuroSys 2020, April 27-30, 2020*, pages 14:1–14:16. ACM, 2020.

[6] Ismail Ari, Ahmed Amer, Robert B. Gramacy, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. ACME: Adaptive Caching Using Multiple Experts. In *Distributed Data & Structures 4, Records of the 4th International Meeting, WDAS 2002, March 20-23, 2002*, Proceedings in Informatics. Carleton Scientific, 2002.

[7] Martin F. Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating Content Management Techniques for Web Proxy Caches. *SIGMETRICS Performance Evaluation Review*, 27(4):3–11, 2000.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, Technical Report UCB/EECS, 28, EECS Department, University of California, Berkeley, 2009.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 12, June 11-15, 2012*, pages 53–64. ACM, 2012.

[10] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *the 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, April 9-11, 2018*, pages 389–403. USENIX Association, 2018.

[11] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, July 12-14, 2017*, pages 499–511. USENIX Association, 2017.

[12] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *the 1st USENIX Symposium on Internet Technologies and Systems, USITS 97, December 8-11, 1997*. USENIX, 1997.

[13] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic Cloud Caching. In *the 7th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 15, July 6-7, 2015*. USENIX Association, 2015.

[14] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, March 16-18, 2016*, pages 379–392. USENIX Association, 2016.

[15] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: A Dynamic Multi-Tenant Key-value Cache. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, July 12-14, 2017*, pages 321–334. USENIX Association, 2017.

[16] Google Cloud. Tulip: Building a Smarter, Leading-Edge Retail Platform. https://cloud.google.com/customers/tulip.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *the 1st ACM Symposium on Cloud Computing, SoCC 2010, June 10-11, 2010*, pages 143–154. ACM, 2010.

[18] Intel Corporation. Write Combining Memory Implementation Guidelines. https://download.intel.com/design/PentiumII/applnots/24442201.pdf.

[19] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[20] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, July 10-12, 2019*, pages 1–14. USENIX Association, 2019.

[21] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage, TOS 2017*, 13(4):35:1–35:31, 2017.

[22] Amazon ElastiCache. https://aws.amazon.com/elasticache/?nc1=h_ls.

[23] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, June 12-16, 2011*, pages 301–312. ACM, 2011.

[24] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen I. Kat. It's Time to Revisit LRU vs. FIFO. In *the 12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*. USENIX Association, 2020.

[25] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, April 2-5, 2013*, pages 371–384. USENIX Association, 2013.

[26] Abraham Flaxman, Adam Tauman Kalai, and H. Brendan McMahan. Online Convex Optimization in the Bandit Setting: Gradient Descent without a Gradient. In *the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, January 23-25, 2005*, pages 385–394. SIAM, 2005.

[27] Dylan J. Foster, Alexander Rakhlin, and Karthik Sridharan. Adaptive Online Learning. *CoRR*, abs/1508.05170, 2015.

[28] Google. Memorystore. https://cloud.google.com/memorystore, 2023.

[29] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *the 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, March 27-29, 2017*, pages 649–667. USENIX Association, 2017.

[30] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A Hardware-Software Co-designed Disaggregated Memory System. In *the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2022, Feb. 28 - Mar. 4, 2022*, pages 417–433. ACM, 2022.

[31] InfiniBand. https://www.infinibandta.org/.

[32] Song Jiang and Xiaodong Zhang. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *the 2002 ACM SIGMETRICS International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, June 15-19, 2002*, pages 31–42. ACM, 2002.

[33] Theodore Johnson and Dennis E. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *the 20th International Conference on Very Large Data Bases, VLDB 1994, September 12-15, 1994*, pages 439–450. Morgan Kaufmann, 1994.

[34] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *ACM SIGCOMM 2014 Conference, SIGCOMM 2014, August 17-22, 2014*, pages 295–306. ACM, 2014.

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.

[36] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, November 2-4, 2016*, pages 185–201. USENIX Association, 2016.

[37] Ricardo Koller and Raju Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *the 8th USENIX Conference on File and Storage Technologies, FAST 2010, February 23-26, 2010*, pages 211–224. USENIX, 2010.

[38] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast Migration for Low-Latency In-Memory Storage. In *the 26th Symposium on Operating Systems Principles, SOSP 2017, October 28-31, 2017*, pages 390–405. ACM, 2017.

[39] Michael Labib. Amazon ElastiCache Deep Dive. https://pages.awscloud.com/rs/112-TZM-766/images/Session%201%20-%20ElastiCache-DeepDive_v2_rev.pdf.

[40] Se Kwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory. *Proceedings of the VLDB Endowment*, 15(13):4023–4037, 2022.

[41] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-Network Memory Management for Disaggregated Data Centers. In *the 28th Symposium on Operating Systems Principles, SOSP 2021, October 26-29, 2021*, pages 488–504. ACM, 2021.

[42] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A Scalable RDMA-Oriented Learned Key-Value Store for Disaggregated Memory Systems. In *the 21st USENIX Conference on File and Storage Technologies, FAST 2023, February 21-23, 2023*, pages 99–114. USENIX Association, 2023.

[43] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, April 2-4, 2014*, pages 429–444. USENIX Association, 2014.

[44] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *the 36th International Symposium on Computer Architecture, ISCA 2009, June 20-24, 2009*, pages 267–278. ACM, 2009.

[45] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *the 17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, July 10-12, 2023*, pages 553–571. USENIX Association, 2023.

[46] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *the 20th USENIX Conference on File and Storage Technologies, FAST 2022, February 22-24, 2022*, pages 313–328. USENIX Association, 2022.

[47] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *the 2nd USENIX Conference on File and Storage Technologies, FAST 2003, March 31 - April 2, 2003*. USENIX, 2003.

[48] Memcached. http://memcached.org, 2022.

[49] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference, USENIX ATC 2013, June 26-28, 2013*, pages 103–114. USENIX Association, 2013.

[50] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, April 2-5, 2013*, pages 385–398. USENIX Association, 2013.

[51] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to Zombieland: Practical and Energy-Efficient Memory Disaggregation in a Datacenter. In *the 13th EuroSys Conference, EuroSys 2018, April 23-26, 2018*, pages 16:1–16:12. ACM, 2018.

[52] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD 1993, May 26-28, 1993*, pages 297–306. ACM Press, 1993.

[53] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *the 19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 217–231. USENIX Association, 2021.

[54] Stefan Podlipnig and László Böszörményi. A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys*, 35(4):374–398, 2003.

[55] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. FairRide: Near-Optimal, Fair Cache Sharing. In *the 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, March 16-18, 2016*, pages 393–406. USENIX Association, 2016.

[56] Xiulei Qin, Wenbo Zhang, Wei Wang, Jun Wei, Xin Zhao, and Tao Huang. Optimizing Data Migration for Cloud-Based Key-Value Stores. In *the 21st ACM International Conference on Information and Knowledge Management, CIKM 2012, October 29 - November 02, 2012*, pages 2204–2208. ACM, 2012.

[57] M. Rajashekhar and Y. Yue. Twemcache. https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache, 2012.

[58] Redis. http://redis.io, 2022.

[59] Liana V. Rodriguez, Farzana Beente Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *the 19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 341–354. USENIX Association, 2021.

[60] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.

[61] Trausti Saemundsson, Hjörtur Björnsson, Gregory V. Chockler, and Ymir Vigfusson. Dynamic Performance Profiling of Cloud Caches. In *the 5th ACM Symposium on Cloud Computing, SoCC 2014, November 3-5, 2014*, pages 28:1–28:14. ACM, 2014.

[62] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 205–218. USENIX Association, 2020.

[63] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.

[64] Yizhou Shan, Will Lin, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. Disaggregating and Consolidating Network Functionalities with SuperNIC. *CoRR*, 2021.

[65] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *the 21st USENIX Conference on File and Storage Technologies, FAST 2023, February 21-23, 2023*, pages 81–98. USENIX Association, 2023.

[66] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *ACM SIGCOMM 2021 Conference, SIGCOMM 2021, August 23-27, 2021*, pages 93–105. ACM, 2021.

[67] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning Relaxed Belady for Content Distribution Network Caching. In *the 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, February 25-27, 2020*, pages 529–544. USENIX Association, 2020.

[68] CXL Specification. https://www.computeexpresslink.org/.

[69] P. Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, May 4-6, 2015*, pages 513–527. USENIX Association, 2015.

[70] Gen-Z Technology. https://genzconsortium.org/.

[71] The Pokemon Company Migrates to AWS Purpose-Built Databases. https://aws.amazon.com/solutions/case-studies/the-pokemon-company-case-study.

[72] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 33–48. USENIX Association, 2020.

[73] Athena Vakali. LRU-Based Algorithms for Web Cache Replacement. In *the 1st International Conference of Electronic Commerce and Web Technologies, EC-Web 2000, September 4-6, 2000*, volume 1875 of *Lecture Notes in Computer Science*, pages 409–418. Springer, 2000.

[74] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving Cache Replacement with ML-Based LeCaR. In *the 10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, July 9-10, 2018*. USENIX Association, 2018.

[75] Carl A. Waldspurger, Nohhyun Park, Alexander T. Garthwaite, and Irfan Ahmad. Efficient MRC Construction with SHARDS. In *the 13th USENIX Conference on File and Storage Technologies, FAST 2015, February 16-19, 2015*, pages 95–110. USENIX Association, 2015.

[76] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated Managed Runtime. In *the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, November 4-6, 2020*, pages 261–280. USENIX Association, 2020.

[77] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *the 2022 ACM SIGMOD/PODS International Conference on Management of Data, SIGMOD 2022, June 12-17, 2022*, pages 1033–1048. ACM, 2022.

[78] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *the 19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 277–292. USENIX Association, 2021.

[79] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-Based Ordered Key-Value Store using Remote Learned Cache. In *the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, November 4-6, 2020*, pages 117–135. USENIX Association, 2020.

[80] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better! In *the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, October 8-10, 2018*, pages 233–251. USENIX Association, 2018.

[81] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *the 7th Symposium on Operating Systems Design and Implementation, OSDI 2006, November 6-8, 2006*, pages 307–320. USENIX Association, 2006.

[82] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *the 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, April 4-6, 2022*, pages 945–960. USENIX Association, 2022.

[83] Juncheng Yang, Yao Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, November 4-6, 2020*, pages 191–208. USENIX Association, 2020.

[84] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: A Memory-Efficient and Scalable In-Memory Key-Value Cache for Small Objects. In *the 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 503–518. USENIX Association, 2021.

[85] Farzana Beente Yusuf, Vitalii Stebliankin, Giuseppe Vietri, and Giri Narasimhan. Cache Replacement as a MAB with Delayed Feedback and Decaying Costs. *arXiv preprint*, 2020.

[86] Junbiao Zhang, Rauf Izmailov, Daniel Reininger, and Maximilian Ott. Web Caching Framework: Analytical Models and Beyond. In *Proceedings 1999 IEEE Workshop on Internet Applications*, pages 132–141. IEEE, 1999.

[87] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast One-Sided RDMA-Based Distributed Transactions for Disaggregated Persistent Memory. In *the 20th USENIX Conference on File and Storage Technologies, FAST 2022, February 22-24*, pages 51–68. USENIX Association, 2022.

[88] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-Sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 15–29. USENIX Association, 2021.