AdaCache: A Disaggregated Cache System with Adaptive Block Size for Cloud Block Storage

Qirui Yang
Samsung
qirui.y@samsung.com

Runyu Jin
Arizona State University
runyu.jin@asu.edu

Ni Fan, Devasena Inupakutika, Bridget Davis
Samsung
ni.fan, devasena.i, b.davis@samsung.com

Ming Zhao
Arizona State University
mingzhao@asu.edu

***Abstract*—NVMe SSD caching has demonstrated impressive capabilities in solving cloud block storage's I/O bottleneck and enhancing application performance in public, private, and hybrid cloud environments. However, traditional host-side caching solutions have several serious limitations. First, the cache cannot be shared across hosts, leading to low cache utilization. Second, the commonly-used fix-sized cache block allocation mechanism is unable to provide good cache performance with low memory overhead for diverse cloud workloads with vastly different I/O patterns. This paper presents AdaCache, a novel userspace disaggregated cache system that utilizes adaptive cache block allocation for cloud block storage. First, AdaCache proposes an innovative adaptive cache block allocation scheme that allocates cache blocks based on the request size to achieve both good cache performance and low memory overhead. Second, AdaCache proposes a group-based cache organization that stores cache blocks into groups to solve the fragmentation problem brought by variable-sized cache blocks. Third, AdaCache designs a two-level cache replacement policy that replaces cache blocks in both single blocks and groups to improve the hit ratio. Experimental results with real-world traces show that AdaCache can substantially improve I/O performance and reduce storage access caused by cache miss with a much lower memory usage compared to traditional fix-sized cache systems.**

***Index Terms*—SSD cache, disaggregated cache, cloud block storage, rack scale disaggregation, NVMeoF**

I. Introduction

Block storage is widely used in public, private, and hybrid cloud environments because it is highly effective in providing fast, scalable, and reliable access to data [1]–[5]. Although cloud block storage is generally considered to be more I/O performant than other types of cloud storage such as object storage and file storage [6], it still falls short of the performance provided by directly attached NVMe SSD storage. To accelerate modern data-intensive applications such as Deep Learning (DL) training and big data processing [7], NVMe SSD caching is employed to exploit workload locality for faster data accesses [8]–[11]. Typically, NVMe SSD cache devices are directly attached to each computing server which is usually multiple network hops away from storage servers [12]. However, this host-side caching mechanism [10], [11] can lead to uneven cache utilization for two reasons. First, different computing servers run different cloud workloads can require varying degrees of cache resources. Second, a cache device is only used by the server where it is attached and cannot be shared or utilized across multiple computing servers.

By decoupling cache devices from computing servers, rack-scale cache disaggregation enables cache sharing through the pooling of cache resources within the same group of racks. Cache resources are managed and allocated as a whole which can lead to better cache utilization, scalability, and failure isolation. To achieve this, NVMe over Fabrics (NVMeoF) [13] can be employed to deliver high performance and scalability. NVMeoF defines a standard protocol for efficiently transporting the NVMe storage protocol over the network, which can scale out to large numbers of NVMe devices and extend the distance over which they can be accessed with low latency and high IOPS within a data center [14].

The fix-sized cache block management method commonly used in various cache system designs may not be the desirable solution for cloud workloads that are constantly changing. Using smaller cache blocks like 32KiB can achieve better I/O performance as it incurs smaller cache miss penalty [15] compared to larger cache block sizes. However, its metadata overhead for managing the cache resource is higher, which causes larger memory footprint as the metadata usually needs to be cached in memory for performance. Conversely, using larger cache blocks such as 512KiB can improve the cache hit ratio [16] by exploiting the spatial locality within the requests and reduce the memory overhead associated with metadata. However, this comes at the cost of larger cache miss penalty, which can significantly reduce I/O performance if the spatial locality is rare.

In this paper, we aim to design a rack-scale disaggregated cache solution that provides good cache performance with low metadata overhead, regardless of the cloud workloads. We propose AdaCache, a rack-scale disaggregated cache system that employs variable-sized cache blocks to adapt to various cloud workloads. AdaCache allocates cache blocks of different sizes based on the I/O request size. For requests with large I/O sizes, large cache blocks are allocated to reduce the number of allocated cache blocks, thus improving I/O performance and reducing metadata memory overhead. For requests with smaller sizes, AdaCache assigns small cache blocks to avoid read/write amplification between the cache system and back-end storage as well as cache pollution.

The contributions of this paper are as follows:

1) The design and implementation of AdaCache, a practi-

Partly supported by National Science Foundation 2126291, 1955593

cal rack-scale disaggregated cache system implemented using the SPDK framework [17] for cloud block storage.
2) The design of adaptive cache block allocation which incorporates three core ideas: efficient variable-sized cache block allocation algorithm, group-based cache organization, and two-level cache replacement.
3) The comprehensive evaluation of AdaCache using publicly available real-world block I/O traces through both simulation and AdaCache prototype.

According to the evaluation results, AdaCache has demonstrated significant improvements in I/O performance compared to traditional fix-sized cache. Specifically, it can improve read latency by 20% and write latency by 9% compared to 32KiB block-sized cache in trace replay. AdaCache is also capable of saving up to 74% I/O traffic to cloud block storage and up to 63% I/O traffic to the cache compared to 256KiB block-sized cache. Moreover, AdaCache has achieved up to 41% memory savings compared to 32KiB block-sized cache. All of these improvements are accomplished with merely 2 microseconds of computation overhead at cache layer compared to a traditional fix-sized cache.

The rest of this paper is structured as follows. In Section II, we introduce the design and implementation of disaggregated cache. In Section III, we elaborate on the details of our AdaCache design. In Section IV, we present our experimental method and the results. In Section V, we discuss the related works and conclude in Section VI.

II. Disaggregated Cache

A. Rack-Scale Cache Disaggregation

Cloud block storage has been widely adopted by today's public, private, and hybrid cloud infrastructure for primary data storage [1]–[4]. With block storage, data is partitioned into fix-sized blocks and stored on the underlying storage medium. These blocks can be directly accessed by applications or through mounted file systems [18], [19], allowing for quick modification of specific blocks to efficiently serve I/O requests.

NVMe SSDs are commonly used as a caching solution in large-scale cloud block storage systems to improve I/O performance [20]. Typically, caches are deployed on computing hosts to mitigate the high network latency to the storage clusters. However, cloud providers often encounter the challenge of load imbalance where some cache devices are more heavily used than others, leading to overloaded, under-loaded, or well-loaded cache devices on computing hosts [21]. This results in unbalanced cache utilization and wasted cache resources.

Cache disaggregation presents a solution to the aforementioned issues by disaggregating all the cache resources, enabling cache to be shared and managed as a whole. It decouples SSD cache from the computing nodes and allows independent utilization of cache resources regardless of where an application is placed. In this sense, the cache resources are shared by all the applications and the cache load imbalance problem is addressed. In cloud environments, this can be achieved at either cluster scale or rack scale. Cluster-scale

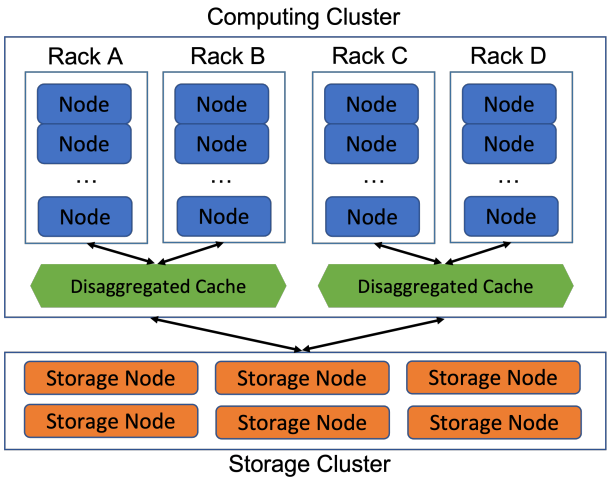


Fig. 1: Rack-Scale Cache Disaggregation

cache disaggregation offers more pooled cache resources and consequently can result in better cache utilization compared to rack-scale. However, it suffers from higher network latency to access cache across the cluster which can negatively impact I/O performance. Additionally, it requires complicated software design and may inversely bring unacceptable software overhead and offset its benefit. Conversely, rack-scale cache disaggregation can provide superior cache resource utilization compared to the local cache and involve much lower network and software overhead compared to cluster-scale. As such, it provides an optimal trade-off between cache resource utilization and I/O performance. Figure 1 illustrates an example of rack-scale cache disaggregation.

Rack-scale cache disaggregation enables cache devices within the same group of racks to share a cache server, providing computing servers of the same rack group with a pool of shared cache resources. The fast data transfer between computing nodes and the cache server can be achieved with the adoption of NVMe over Fabrics (NVMeoF) [13] technology, which is a protocol designed to provide storage to computing servers through the network using the NVMe protocol. It adds less than 10 microseconds of additional latency to locally attached NVMe devices [22], making it an ideal choice for connecting the cache pool to the computing nodes. According to a recent performance report [23], NVMeoF using RDMA [13] has demonstrated impressive speed, achieving more than 11M 4K IOPS with an average latency of 231 microseconds using 100 Gbps NICs. As network bandwidth continues to double every few years, this performance is expected to improve even further. With such high performance, a single cache server can effectively serve thousands of concurrent NVMeoF connections. Furthermore, a single cache server can provide large storage capacities. For example, Samsung's Poseidon reference system [24] can support up to 24 Samsung PM1733 NVMe SSDs with a total capacity of up to 368TiB. This capacity is sufficient to support thousands of cache clients for cloud block storage.

Figure 2 compares the I/O performance of different storage

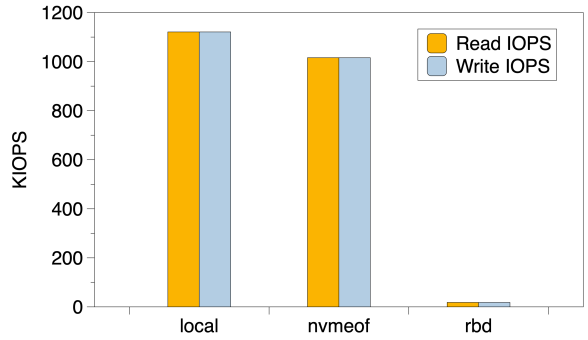


Fig. 2: IOPS Comparison of Local SSDs, NVMeoF SSDs, and All-Flash Ceph RBD.

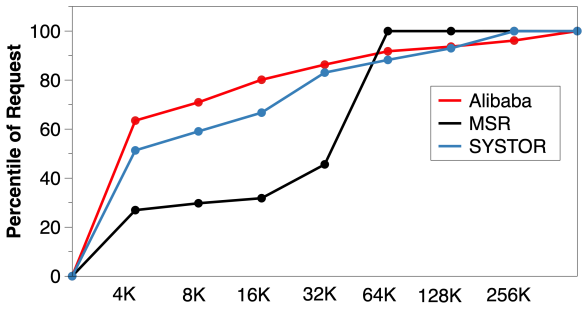


Fig. 3: Request Size CDF of different traces

setup: local NVMe SSDs (local), remote NVMeoF SSDs (nvmeof), and remote all-flash Ceph Rados Block Devices (rbd) [4]. Local and nvmeof each consists of four Samsung PM9A3 NVMe SSDs that form a RAID0. Rbd consists of 12 Samsung PM9A3 NVMe SSDs from a 3-node Ceph cluster that form a RAID0. We use local to demonstrate the performance of the local cache, and nvmeof to demonstrate the performance of the disaggregated cache. Rbd is an open-sourced cloud block storage system used to demonstrate the performance of cloud block storage without NVMe SSD caching. We ran the FIO [25] benchmark issuing 30 minutes of asynchronous random 4K reads and writes with the same I/O queue depth to different storage setups. We observe that local NVMe SSDs outperform cloud block storage by 60X. Remote SSDs using NVMeoF have comparable performance to local NVMe SSDs with merely a 9% drop in IOPS.

B. Rack-Scale Cache Management

A cache block is the minimum unit of cache that can be read from or written to. The block size determines the size of an I/O operation that can be performed. Common cache block sizes range from 512B to 64KiB [8], [10], [11], [26], [27]. The choice of cache block size can impact the performance, endurance, and cost of a storage solution by affecting cache hit ratio, I/O volume, and in-memory metadata overhead. Therefore, it's important to select a cache block size that fits the workload best. Smaller cache blocks often have better I/O performance due to the smaller I/O volume, which comes from the smaller cache block allocation and smaller cache miss penalty. However, they may have a lower cache hit ratio because they cannot fully leverage the spatial locality within the application requests [16].

For a rack-scale cache with hundreds of terabytes of cache space, the large memory footprint for the metadata is another concern for small block sizes. For example, assuming each cache block only requires 40 bytes of memory metadata to provide a source address to cache address mapping (including source address, cache address, a pointer for indexing, and two pointers for LRU) [10], [28], a 368 TiB cache with 16 KiB cache block size would require 920 GiB of memory footprint, which is difficult to fit in memory, considering memory density grows 10 times slower than SSD density [29].

Large cache blocks, on the other hand, can potentially improve hit ratio [16] due to better exploitation of spatial locality. Additionally, the memory footprint reduces linearly with the increased size of the cache block. Take the last example: a 368 TiB cache with 512 KiB cache block size would require merely 29 GiB of memory footprint. However, large cache blocks lead to large cache block allocation and large cache miss penalty which can significantly harm I/O performance. These reasons stop large cache blocks from being applied in reality. Section IV presents a thorough comparison of I/O performance using cache of different cache block sizes.

The cloud environment is dynamic and changes rapidly over time with varying workloads. Some workloads involve small requests, such as those from transactional databases, while others have large requests, such as those from multimedia systems. We conducted an analysis of request size cumulative distribution functions (CDF) from three real-world traces: Alibaba block I/O Traces (*alibaba*) [30], MSR Cambridge Traces (*msr*) [31], and Systor '17 Traces (*systor*) [32] (detailed information about the traces is presented in Section IV). Figure 3 shows the results. We observe that the distribution of request sizes varies across the traces. For *alibaba* and *systor*, more than half of the requests are smaller than or equal to 4KiB. For *msr*, more than half of the requests are larger than 32KiB. Based on the above observations, a traditional fix-sized block cache is insufficient for today's complex cloud environment. Instead, we design an adaptive cache that can adapt the cache block size to different cloud workloads which is elaborated in Section III.

C. Implementation

AdaCache extends PoseidonOS [33], a userspace software-defined storage (SDS) solution providing high-throughput and low-latency flash storage virtualization with capacity elasticity and data protection (RAID), to offer rack-scale disaggregated cache service for cloud block storage. It is implemented as a virtual block device (bdev) module [34] using the SPDK framework. By using a virtual bdev module, AdaCache can be seamlessly integrated with a wide range of cloud block storage bdevs, enabling compatibility with existing storage systems.

Figure 4 illustrates the architecture of AdaCache. Each local NVMe SSD is represented by a cache bdev in the SPDK

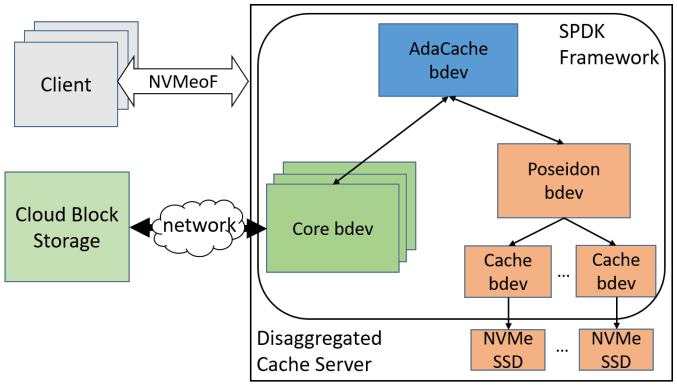


Fig. 4: Disaggregated Cache Architecture

framework. All the cache bdevs are managed by PoseidonOS to offer a large virtualized disaggregated cache space to AdaCache. Each virtual drive in the cloud block storage is represented by a core bdev. AdaCache claims the cache and core bdevs and redirects I/Os between them with no requirement for knowledge of the I/O and network protocol specifics of the underlying bdevs. AdaCache uses GLib's [35] hash table implementation for the in-memory key-value stores.

III. Adaptive Cache Block Size

A. Fix-sized Cache Allocation

Traditional fix-sized cache block allocation has three major steps: address alignment, address lookup, and cache block allocation. Address alignment aligns the offset of the original I/O requests to the aligned offset based on cache block size. Assume R_o is the request offset, B is the cache block size, and A_o is the aligned offset. A_o is computed using the following Equation 1.

$$A_o = floor(R_o/B) * B \quad (1)$$

For example, a read request with offset 33KiB using 32KiB as cache block size aligns to aligned offset 32KiB.

During address lookup, the aligned offset is used as the key to look up the cache address in an in-memory key-value store. In case of a read cache hit, data is read from the cache address directly. Otherwise, a new cache block is allocated and data is read from the backend storage and cached to the newly allocated cache block.

In case of a write cache miss, data is first read from the backend storage and cached to a newly allocated cache block. If the cache uses write-back policy, data is written to the cache block and dirty cache blocks are written back to the backend storage periodically or when they are replaced from the cache. If the cache uses write-through policy, data is written to the cache block and backend storage simultaneously to maintain data consistency. When the cache becomes full, a replacement algorithm such as Least Recently Used (LRU) or Least Frequently Used (LFU) is used to determine which data to replace before allocation happens.

B. Variable-Sized Cache Allocation

Cloud workloads are dynamic in nature, and therefore, the cache system should be able to adapt itself to different workloads that may have varying request sizes. For small requests, small cache blocks are deemed sufficient while large cache blocks may cache unnecessary data, resulting in cache pollution and increased I/O volumes. Conversely, for large requests, large cache blocks can reduce the number of I/Os between the cache and the cloud block storage, and can also reduce the metadata memory overhead. AdaCache uses adaptive cache block allocation which allocates different sizes of cache blocks based on the request size.

AdaCache first generates a list of missing intervals for all the parts of the request that are missing in the cache. As shown in Figure 3, a request can be larger than 256KiB and cover multiple cache blocks. AdaCache determines the aligned range of the request by aligning the request offset and end address (offset + length) to the smallest block size and iterates through the request to find out all the missing intervals.

Because the cache employs variable cache block sizes, it needs to check the in-memory key-value store of every block size to find out if any part of the request is cached under each block size. Figure 5 illustrates an example where a request at offset 48KiB with length 184KiB on a cache that employs cache block sizes of 32KiB, 64KiB, 128KiB, and 256KiB. In this example, the latter part of the request (from 128KiB to 232KiB) is cached under the 128KiB block size. The aligned request range is from 32KiB to 256KiB.

Within the request range, AdaCache starts the search from the smallest cache block size (32KiB in the example), and checks if the current address is cached under any of the cache block sizes. AdaCache first aligns the current address to different cache block sizes using Equation 1. For the example, the aligned offsets are 32KiB, 0, 0, and 0 for the cache block sizes of 32KiB, 64KiB, 128KiB, and 256KiB respectively. It then uses these aligned offsets to search the in-memory key-value store of each cache block size. If the result is all misses, then it knows that the current address with the smallest cache block size (the interval between 32KiB and 64KiB in the example) is not cached, and it adds the interval to the list of missing intervals. AdaCache merges missing intervals if they are contiguous to allocate the largest possible cache block for the intervals. AdaCache then moves on to the next address covered by the request (64KiB in the example) and repeats the above process. After checking the whole request, AdaCache gets a complete list of missing intervals. In the example, the interval from 32KiB to 128KiB is missing in the cache. Algorithm 1 presents the pseudo-code of the missing intervals generation.

For each missing interval in the list, AdaCache tries to allocate using the largest possible cache block size. This greedy allocation ensures that AdaCache reduces the number of allocated cache blocks and I/O counts. To determine if a block size is suitable for the missing interval, AdaCache makes sure the cache block is within the range of the missing intervals

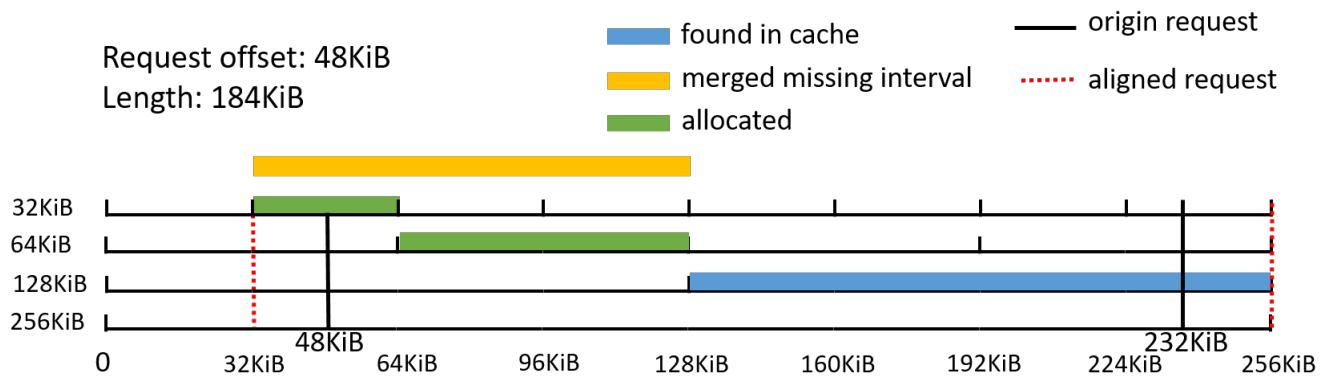


Fig. 5: Adaptive Cache Block Allocation

Algorithm 1 Missing Intervals Generation

Remarks:
  $B_n, \dots B_1$: block size from large to small
  H_B: hash table for block size B
  $A_B(O)$: align offset O using block size B
  $M_{AP}(B, E)$: merge offset interval $\{B, E\}$ to $MissingIntervals$
Inputs:
  O: request offset in bytes
  L: request length in bytes
Output:
  $MissingIntervals$: a list of missing cache blocks
$MissingIntervals \leftarrow \{\}$
$begin \leftarrow A_{B_1}(O)$
$end \leftarrow A_{B_1}(O + L) + B_1$
while $begin \neq end$ **do**
  $hit \leftarrow$ **false**
  for $B \leftarrow B_1, \dots B_n$ **do**
    $begin\_aligned = A_B(begin)$
    if $begin\_aligned \in H_B$ **then**
      $begin \leftarrow begin\_aligned + B$
      $hit \leftarrow$ **true**
      break
    end if
  end for
  if $hit \neq$ **true then**
    $M_{AP}(begin, begin + B_1)$
    $begin \leftarrow begin + B_1$
  end if
end while
return $MissingIntervals$

Algorithm 2 Greedy Cache Block Allocation

Remarks:
  $B_1, \dots B_n$: block size from small to large
  H_B: hash table for block size B
  $A_B(O)$: align offset O on block size B
  $BA(I)$: the begin address of interval I
  $EA(I)$: the end address of interval I
Inputs:
  $MissingIntervals$: a list of cache blocks to allocate
for each $I \in MissingIntervals$ **do**
  $begin \leftarrow BA(I)$
  $end \leftarrow EA(I)$
  while $begin \neq end$ **do**
    for $B \leftarrow B_n, \dots B_1$ **do**
      if $begin \neq A_B(begin)$ **then**
        continue
      end if
      if $B > end - begin$ **then**
        continue
      end if
      $H_B \leftarrow begin \cup H_B$ ▷ allocate cache block
      $begin \leftarrow begin + B$
    end for
  end while
end for

because the addresses that go beyond these intervals may have been cached.

In the example, AdaCache first checks how to allocate for the interval from 32KiB to 128KiB. The largest possible cache block for this interval is actually 32KiB, because all the larger cache blocks start beyond this interval. For the remaining missing interval from 64KiB to 128KiB, the largest possible cache block is 64KiB, because the interval from 64KiB to 128KiB is within the range of the missing interval (64KiB to 128KiB). Therefore, at the end of this greedy allocation process, AdaCache caches two blocks that include one 32KiB cache block from 32KiB to 64KiB and one 64KiB cache block from 64KiB to 128KiB. Algorithm 2 presents the pseudo-code of the greedy cache block allocation.

Assuming N is the request length, M is the number of different cache block sizes, and K is the total number of cache blocks in the cache, the algorithm's time complexity of fix-sized and adaptive cache block allocation have upper bounds of O($K * N$) and O($K * N * M$), respectively. In practice, M is set to a constant value, such as 4 in Figure 5 where the

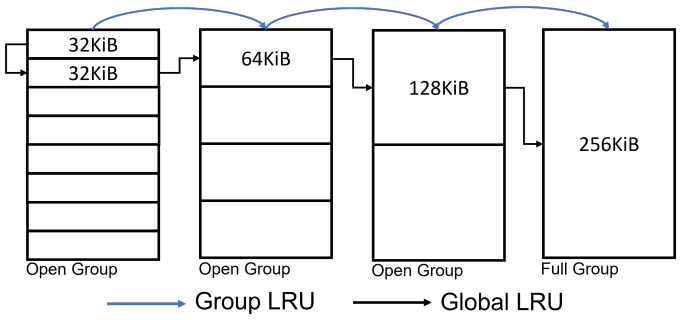


Fig. 6: Group-Based Cache Organization

time complexity can be approximated as O($K * N$), which is equivalent to the fix-sized cache block allocation. The space complexity of the algorithm is identical to the fix-sized cache block allocation, which is O(K).

C. Group-Based Cache Organization

Adaptive cache block allocation is an effective technique that can leverage both small and large blocks, making it suitable for dynamic cloud workloads. However, it incurs fragmentation. When the cache becomes full and adaptive cache blocks get allocated, the cache space is divided into non-contiguous variable-sized pieces. When large requests come, the replacement of smaller blocks can generate many scattered small holes and it is hard to fit a large cache block in.

To address the issue of fragmentation, AdaCache utilizes the concept of slab allocator [28], [36], which involves grouping cache blocks of the same size together into identical-sized groups. Cache blocks belonging to the same group are stored physically adjacent to each other in the cache. Consequently, when the cache is full, a whole group is replaced, creating a contiguous piece of cache space for cache block allocation.

AdaCache chooses the largest cache block size as the group size. In this way, replacement of a whole group can free just enough cache space for the largest cache block allocation. In the case of small block allocation, the replacement of a whole group creates an open group that can be used to allocate many cache blocks of that block size. Figure 6 illustrates an example of the group-based cache organization. The cache block sizes are 32KiB, 64KiB, 128KiB, and 256KiB and the group size is 256KiB. There are three open groups storing 32KiB, 64KiB, and 128KiB cache blocks, respectively, and one full group storing a 256KiB cache block.

When allocating a cache block, AdaCache checks if the cache is full. If it is not, the allocator examines if there is an open group with the same block size. If such a group exists, the block is allocated from the open group. If there is no such open group, AdaCache creates a new one and allocates the cache block from there. If the cache is full, AdaCache replaces an entire group and follows the above procedure. Assume M is the number of different cache block sizes, there are a maximum of M open groups kept in the cache at any given time, and it does not waste significant cache space. For example, in Figure 6, at most 4 256KiB open groups are kept in the cache and used to allocate cache blocks for coming requests.

TABLE I: Specifications Of The Testbed.

Server	CPU	DRAM	SSD	OS	Software
Client	2x Intel Platinum 8260 96 cores	384GB DDR4	/	Ubuntu 18.04	Replayer / Simulator
Disaggregated Cache Server	2x Intel Platinum 8260 96 cores	384GB DDR4	4x Samsung PM9A3 PCIe Gen3 3.84TB	Ubuntu 20.04	Poseidon OS v0.11
3-node Ceph RBD	2x AMD EPYC 7702	512GB DDR4	4x Samsung PM9A3 PCIe Gen4 3.84TB	Ubuntu 20.04	Ceph Quincy

D. Two-Level Cache Replacement

Following group-based cache organization, AdaCache uses a group-based LRU replacement policy that links all the groups together for cache replacement. When a cache block is accessed, the group that contains the cache block is promoted to the head of the group-based LRU list. When the cache is full, AdaCache replaces the group that is at the tail of the LRU list. Although each cache miss may trigger a write-back I/O of the whole group to be evicted, the I/O volume is smaller than that of using large fix-sized cache blocks. Every time a whole group is evicted, all of its space is freed up at once in the cache and can be used to store a number of small cache blocks from future requests.

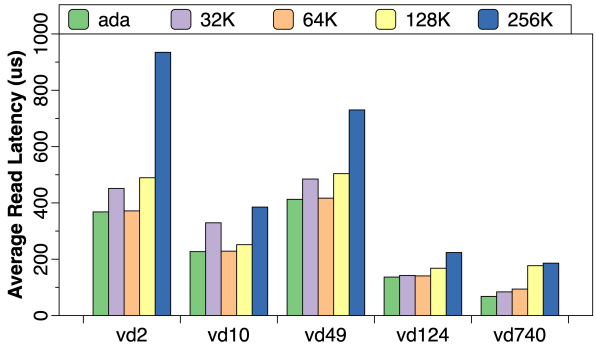
One potential drawback of the group-based replacement policy is that it may retain cold blocks that are in the same group as the frequently accessed hot blocks in the cache, leading to cache pollution. To alleviate the problem, AdaCache incorporates a global cache block LRU replacement policy in addition to the group-based replacement policy. Figure 6 illustrates the two-level LRU lists.

All the cache blocks are linked using a global LRU list. When AdaCache tries to allocate a new cache block in case of a full cache, it first checks the tail of the global LRU list. If the tail cache block has the same size as the new cache block, AdaCache replaces it and promotes both the cache block and its group to the head of the LRU lists. If the size mismatches, AdaCache uses group-based LRU replacement policy to replace a whole group. The use of two-level cache replacement does not incur high lock contention overhead when the cache is accessed in parallel as AdaCache leverages the lockless design of modern high performance storage framework [17].

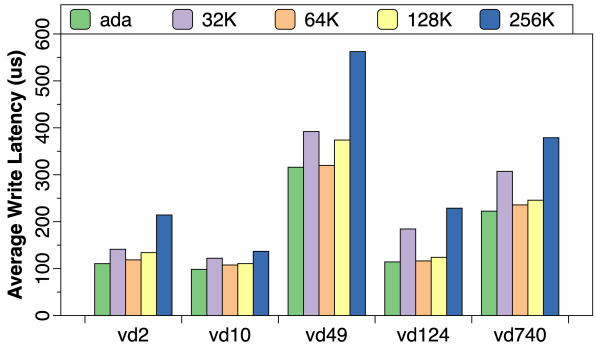
IV. Evaluation

We evaluate the performance of AdaCache using both the simulation and prototype following the design and implementation described in Section III.

Testbed Setup. The testbed consists of three components which are the client, the disaggregated cache server, and the

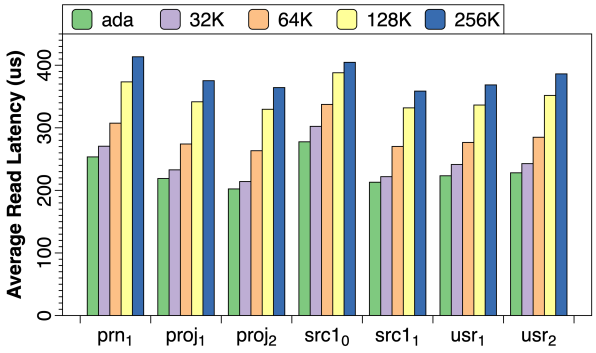


(a) Read Latency

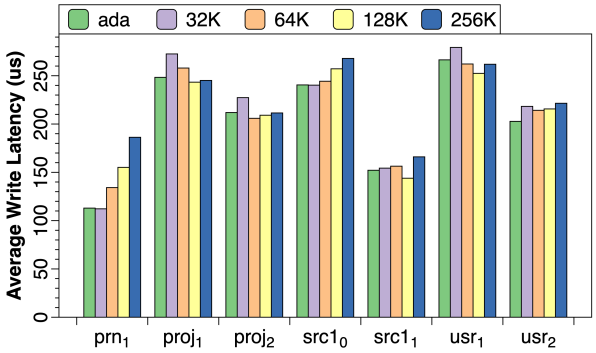


(b) Write Latency

Fig. 7: I/O Latency for Alibaba Trace Replay



(a) Read Latency



(b) Write Latency

Fig. 8: I/O Latency for Msr Trace Replay

TABLE II: Trace Segments Statistics.

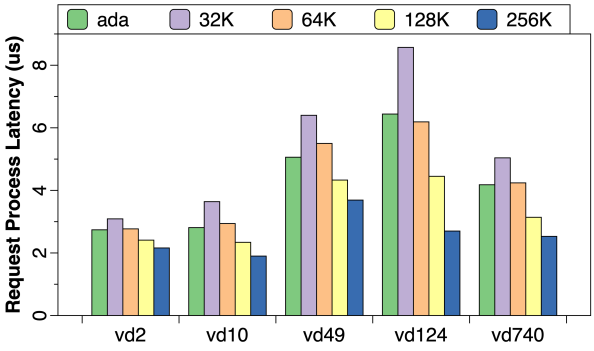
	alibaba	***msr***	***systor***
#Reads	24.5M	61M	40.7M
#Writes	25.5M	9M	19.3M
Read Traffic GiB	607.3	2416.8	1109.2
Write Traffic GiB	375.9	207.2	271.9

cloud block storage cluster. The client issues the I/O workloads to the disaggregated cache server through NVMeoF RDMA using a 100Gbps NIC. The disaggregated cache server runs AdaCache and provides the cloud block storage with NVMe SSD caching through the network using another 100Gbps NIC. The disaggregated cache server is configured as RAID0 using PoseidonOS consisting of four NVMe SSDs. The cloud block storage is a three-node Ceph cluster with Ceph Rados Block Devices (RBDs). The specs for each component are shown in Table I.

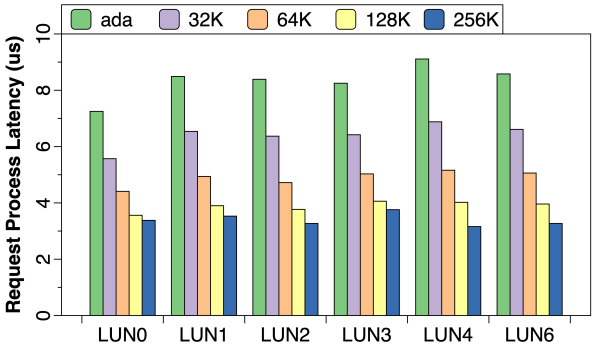
Workloads. We considered the following three real-world block I/O traces to provide a comprehensive evaluation:

- Alibaba block I/O Traces [30] (*alibaba*): *alibaba* is collected from an elastic block service cluster of Alibaba Cloud and it contains I/Os from 1000 virtual disks. Among them, we picked 5 virtual disks (vd2, vd10, vd49, vd124, and vd740) that have a large amount of I/O volumes for trace replay. We replayed the first 10 million I/O requests issued to the 5 virtual disks concurrently. Requests to vd2 and vd740 are write-dominant while I/Os to vd10 and vd124 are read-dominant. Vd49 has a similar amount of read and write I/Os.
- MSR Cambridge Traces [31] (*msr*): *msr* is block-level traces collected from Microsoft Research enterprise data centers and it contains I/Os from 13 servers. Among them, we picked traces from seven drives (prn_1, proj_1, proj_2, src1_0, src1_1, usr_1, and usr_2) that have more than 10 million I/Os. We replayed the first 10 million I/Os issued to the 7 servers concurrently. All the *msr* traces are read-dominant.
- Systor '17 Traces [32] (*systor*): *systor* is collected from an enterprise Virtual Desktop Infrastructure (VDI) which contains I/Os from 300 VMs. All these VMs share 6 storage logical unit numbers (LUN). We replayed the first 10 million I/Os issued to the 6 LUNs concurrently. All the *systor* traces are read-dominant.

For trace segments replay, the cache employs a write-back policy and we can leverage related work [37] to ensure cache consistency. Trace segments are replayed using pread() and pwrite() to issue direct I/Os to different target devices in parallel according to the trace. Each target device consists of 1 TiB Ceph RBD as the backend storage and 10% of each trace's total working set size (WSS) as the cache size. Table II shows the statistics of the trace segments that we use for replay. We also replay the entire traces using a simulator with the same implementation as the AdaCache prototype to show metrics from the whole trace simulation. In the evaluation, the cache block sizes used by AdaCache are 32KiB, 64KiB, 128KiB, and 256KiB. We compare AdaCache to fix-sized disaggregated caches with these four cache block sizes. Each experiment is

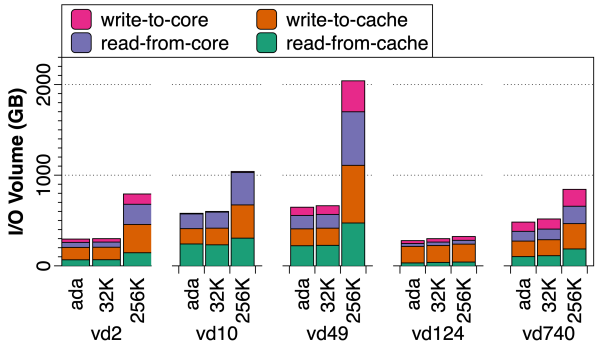


(a) Alibaba Trace Replay

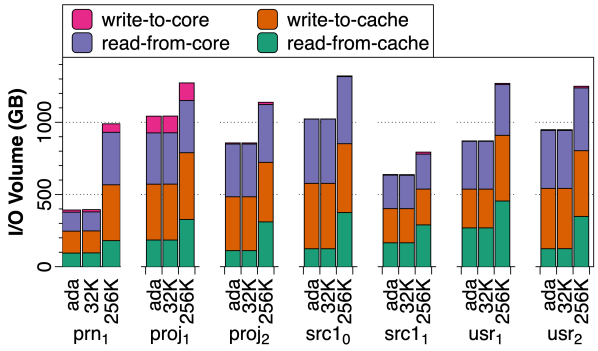


(b) Systor Trace Replay

Fig. 9: Request Processing Latency



(a) Alibaba Trace Replay



(b) Msr Trace Replay

Fig. 10: I/O volumes

repeated three times and we show the average results here. Due to the space limit, we only show evaluation results that are representative of all results.

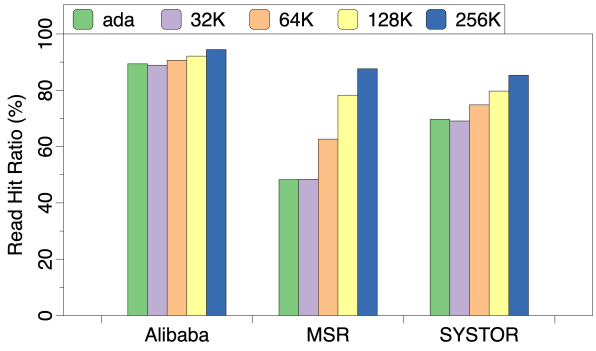
A. *I/O performance*

I/O Latency. Figure 7 shows the average read and write latency from *alibaba* trace replay. Results reveal that AdaCache has the best overall read and write latency compared to fix-sized caches with different trace segments. For read latency, AdaCache improves it by 19% for trace segment vd740 compared to 32KiB cache and 63% compared to 256KiB cache. For write, AdaCache has an improvement of 9% for trace segment vd10 compared to 64KiB cache and 50% for vd124 compared to 256KiB cache. Figure 8 shows the read and write latency from *msr* trace replay. AdaCache also improves the read latency by 7% compared to 32KiB cache for usr_1 and 44% compared to 256KiB cache for proj_2. For write latency, AdaCache can improve it by 9% compared to 32KiB cache for proj_1 and 39% compared to 256KiB cache for prn_1.

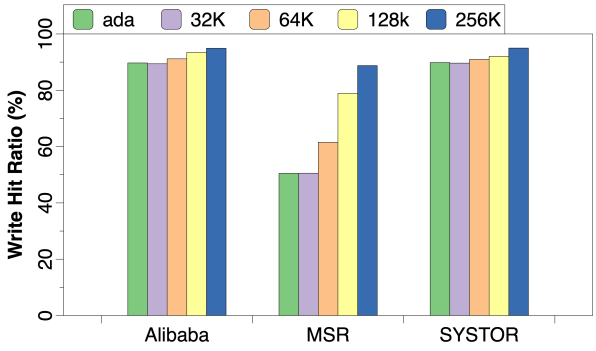
Comparing the two traces' latency results from fix-sized caches, *alibaba* mostly has the best read and write performance when using a 64KiB cache. *Msr* has the best read performance when using a 32KiB cache. For write, different cache block sizes perform differently for different trace segments. For example, trace segment prn_1 has the best write performance using 32KiB cache while trace segment proj_1 performs the best using 128KiB cache. This also proves that a fix-sized cache cannot provide optimal performance for different cloud workloads. Of the two traces, AdaCache outperforms all the fix-sized caches in both read and write. Although AdaCache has similar I/O volumes as 32KiB cache (discussed later in Section IV-B), it is achieving better performance because of the adaptiveness of AdaCache which allocates large cache blocks for large requests. These large cache blocks have reduced the number of I/Os and can therefore improve the performance.

Average Request Processing Latency. Figure 9 shows the average request processing latency from trace replay. This latency is captured from when an I/O request is received by the cache to when a processed I/O request is sent to the storage devices. It includes the latency for the cache block allocation as described in Section III-A and III-B. This illustrates the cache block allocation overhead of AdaCache compared to fix-sized caches. Figure 9a shows the request processing latency from *alibaba* trace replay. For fix-sized caches, large cache blocks can reduce the number of cache block allocations and therefore reduce the request processing latency. We also observe that AdaCache outperforms 32KiB cache in request processing latency by 25% for vd124. There are two reasons behind this. First, AdaCache uses large cache blocks for large requests which can help reduce the average request processing latency. Second, the high hit ratio for *alibaba* trace segment (around 70% for read and 90% for write) has amortized the extra overhead of adaptive cache block allocation.

Figure 9b shows the results from *systor* trace replay. We observe that AdaCache has larger average request process latency than fix-sized caches by 29% compared to 32KiB



(a) Read Hit Ratio



(b) Write Hit Ratio

Fig. 11: Whole Trace simulation results

cache for LUN1. *Systor* trace segment has around 60% read hit ratio and because it is read dominant, the low hit ratio fails to amortize the overhead. Although AdaCache brings extra process overhead from adaptive cache block allocation, the overhead is merely a few microseconds and does not hurt the I/O performance as we have seen previously from the I/O latency results.

B. I/O Volumes

Figure 10 shows the total I/O volumes from *alibaba* trace replay and *msr* trace replay. The I/O volume consists of writes to the cloud block storage (write-to-core), reads from the cloud block storage (read-from-core), writes to the cache (write-to-cache), and reads from the cache (reads-from-cache). Due to the space limit, we only show 32KiB cache and 256KiB cache I/O volumes which have the smallest and the largest amount of I/O volumes, respectively. As discussed in Section II, using large cache blocks may cache unnecessary data and lead to cache pollution and high cache miss penalty. We also observe that AdaCache has a similar amount of I/O volumes as the 32KiB cache. This is because although it uses large cache blocks, it caches only necessary data based on the request size. It does not suffer from the large cache miss penalty as the 256KiB cache does. Of the four types of I/O volumes, I/Os to cloud block storage has much larger overhead than I/Os to cache. Compared to 256KiB cache, AdaCache can save 74% I/Os to cloud block storage and 63% I/Os to cache for vd49 from *alibaba*.

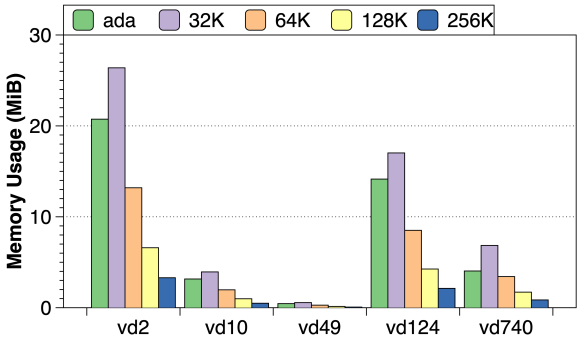


Fig. 12: Memory Usage For Alibaba Trace Replay

C. Memory Usage

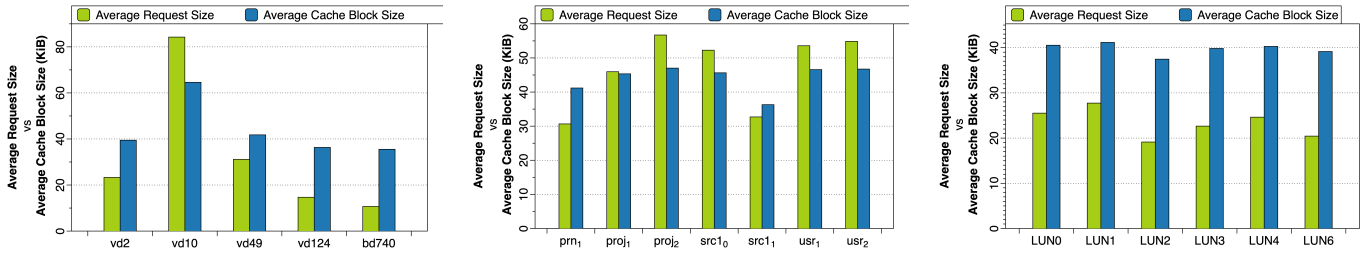
Figure 12 compares the average metadata memory usage of AdaCache to fix-sized caches during the trace replay of *alibaba*. For larger cache blocks, the number of cache blocks used is smaller which leads to smaller metadata memory usage. AdaCache saves 41% memory usage compared to 32KiB cache for vd740. From the request size analysis in Section II, *alibaba* mostly consists of small requests. For workloads that have larger requests, AdaCache tends to allocate larger cache blocks and can potentially save more memory.

D. Hit Ratio

Figure 11 shows the read and write hit ratio from the whole trace simulation of *alibaba*, *msr*, and *systor*. As discussed in Section II, larger cache blocks can benefit from the potential spatial locality within the requests and can achieve better hit ratio compared to smaller cache blocks. We also observe similar behavior when replaying the trace segments. For the whole trace simulation, compared to 256KiB cache, AdaCache has up to 39% drop in read hit ratio and up to 38% drop in write hit ratio from *msr*. For trace replay, AdaCache has up to 60% drop in read hit ratio and up to 59% drop in write hit ratio from *msr* compared to 256KiB cache. Although the hit ratio is much lower for AdaCache, it has up to 39% improvement in write performance and 40% improvement in read performance in trace replay compared to 256KiB cache. This shows that compared to the hit ratio and memory usage, I/O volumes play a more significant role in affecting the cache performance.

E. Effectiveness of Adaptive Cache Block Allocation

Figure 13 validates the effectiveness of AdaCache block allocation algorithms. It shows two metrics: the average request size for all the missed requests v.s. the average cache block size that AdaCache allocates when a cache miss occurs during trace replay. The core design idea of AdaCache is to adaptively allocate variable-sized cache blocks based on the request size. The differences between these two metrics tell us how well AdaCache follows the design idea. We observe that AdaCache follows the trend of the request size to allocate cache blocks. With larger requests, the average cache block size also gets larger. For small requests which are mostly seen from *alibaba* and *systor*, the average cache block size of



(a) Alibaba Trace Replay (b) Msr Trace Replay (c) Systor Trace Replay

Fig. 13: Average Request Size v.s Average Cache Block Size

AdaCache is bounded by the smallest cache block size 32KiB. For the best case, AdaCache achieves merely a 1% difference in *msr* trace replay of trace segment proj_1.

V. Related Works

Flash Caching. Flash caching [38]–[41] has been extensively studied to improve the I/O performance for slow primary storage systems. Solutions have been proposed to solve the capacity and endurance [11], [42], multi-tenancy [8], [10], [26], [41], [43] and multi-tier [44]–[46] problems of flash caching. For example, CloudCache [10] presents an on-demand cache management solution that meets the performance requirements of each tenant by introducing the Reuse Working Set (RWS) cache demand model. SHARDS [26] is an Miss Ratio Curve (MRC) approximation algorithm that focuses on improving MRC efficiency for online cache reassignment by employing uniform randomized spatial sampling. These orthogonal works can be integrated with AdaCache to improve the cache utilization in a disaggregated cloud environment. Nitro [42] is a host-side flash cache solution that performs deduplication and compression on the data blocks, after which the compressed variable-sized data chunks are stored in the cache as fixed-size Write-Evict Units (WEUs). Nitro uses LRU at the granularity of WEU for cache replacement. Besides the coarse-grained cache replacement policy employed by both Nitro and AdaCache, AdaCache also uses the fine-grained cache block replacement policy to further improve the cache hit ratio by replacing cold cache blocks inside each group as discussed Section III.

Flash Disaggregation. Storage disaggregation [1], [4], [47]–[50] is common practice in production environment. High-performance flash disaggregation is also an active research area [51], [52]. Since modern NVMe SSDs are significantly faster than SATA SSDs and hard drives, the software overhead becomes nonnegligible. Guz et al. [51] evaluated the overhead of NVMe SSD storage disaggregation through NVMeoF [13] and concluded that the overhead of remote access is negligible compared to local NVMe SSDs. Decibel [52] is a solution for flash storage disaggregation at the rack scale, which follows a design of sharing-nothing and provides virtualized storage with low latency by minimizing the software overhead through the integration of network and storage layers.

In-Memory Caching. In-memory caching systems [28], [53], [54] are widely used in modern software architecture to improve application performance and scalability. For example, Memcached [28] is a lightweight DRAM key-value store that stores key-value pairs of the same value size in slabs of the same slab class. Unlike AdaCache which does global cache block groups replacement, Memcached does time-consuming slab reassignment [55]–[57] across slab classes due to the high concurrency. Data structure optimization [58]–[60] to save the metadata memory overhead has also been studied. For example, MemC3 [58] reduces the metadata memory footprint by up to 30% for Memcached by using concurrent Cuckoo hashing and CLOCK LRU-approximation cache replacement. These data structure optimization techniques are complementary to AdaCache and can be leveraged to further reduce the metadata memory overhead.

Adaptive Cache Block Sizes. The performance impact of varying cache block sizes for both memory and storage cache has been thoroughly studied in literature [61]–[66]. However, few have studied the benefits and drawbacks of a cache system with adaptive cache block sizes. Jeremic et al. [67] proposed a two-size cache block allocation mechanism that employs a small-block and a large-block SSD cache. The source address space is divided into segments of contiguous source blocks where either the small or the large cache block size can be used. The assignment relationship between segments and cache block sizes is adjusted in the background based on the measurement of I/O latency. AdaCache differs from the related work including but not limited to 1) AdaCache supports different numbers of cache block sizes to cater to the workloads' characteristics without delay, 2) AdaCache adapts the cache block size based on the request size which is more efficient and effective than monitoring I/O latency of the system. To our best knowledge, AdaCache is the first practical storage cache solution using adaptive cache block sizes.

VI. Conclusion

This paper presents a cache system optimized for cloud block storage with constantly changing workloads. The novelties of this work lie in a new cache block allocation design that dynamically adapts the cache block size to the workloads' characteristics. The entire work is cautiously designed to solve the challenges brought by variable-sized cache block allocation. An extensive experimental evaluation based on real-world block traces confirms that AdaCache can achieve vast improvements in I/O performance and memory usage with negligible run-time overhead.

REFERENCES

[1] "Amazon elastic block store (ebs)," 2023. https://aws.amazon.com/ebs/.

[2] "Persistent disk," 2023. https://cloud.google.com/persistent-disk.

[3] "Ibm cloud block storage," 2023. https://www.ibm.com/cloud/block-storage.

[4] "Ceph block device," 2023. https://docs.ceph.com/en/quincy/rbd/index.html.

[5] "What is block storage," 2021. https://aws.amazon.com/what-is/block-storage/.

[6] "Ebs pricing and performance: A comparison with amazon efs and amazon s3," 2018. https://bluexp.netapp.com/blog/ebs-efs-amazons3-best-cloud-storage-system.

[7] Q. Yang, R. Jin, B. Davis, D. Inupakutika, and M. Zhao, "Performance evaluation on cxl-enabled hybrid memory pool," in *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–5, 2022.

[8] Y. Zhang, P. Huang, K. Zhou, H. Wang, J. Hu, Y. Ji, and B. Cheng, "Osca: An online-model based cache allocation scheme in cloud block storage systems," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pp. 785–798, 2020.

[9] K. Zhou, Y. Zhang, P. Huang, H. Wang, Y. Ji, B. Cheng, and Y. Liu, "Efficient ssd cache for cloud block storage via leveraging block reuse distances," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2496–2509, 2020.

[10] D. Arteaga, I. Ahmad, J. Cabrera, S. Jun, J. Xu, S. Xu, S. Sundararaman, M. Zhao, S. Zhen, V. Tarasov, *et al.*, "Cloudcache: On-demand flash cache management for cloud computing," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pp. 355–369, 2016.

[11] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "Cachededup: In-line deduplication for flash caching," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pp. 301–314, 2016.

[12] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, *et al.*, "Understanding rack-scale disaggregated storage.," *HotStorage*, vol. 17, p. 2, 2017.

[13] "Nvm express moves into the future," 2023. https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf.

[14] "Making a case for a disaggregated storage architecture," 2021. https://www.kalrayinc.com/blog/making-case-disaggregated-storage-architecture/.

[15] S. Prybylski, M. Horowitz, and J. Hennessy, "Performance tradeoffs in cache design," *SIGARCH Comput. Archit. News*, vol. 16, p. 290–298, may 1988.

[16] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[17] I. Corporation, "SPDK: Storage Performance Development Kit." https://spdk.io/, Accessed 2023.

[18] Y. Liu, H. Li, Y. Lu, Z. Chen, and M. Zhao, "An efficient and flexible metadata management layer for local file systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 208–216, 2019.

[19] Y. Liu, H. Li, Y. Lu, Z. Chen, N. Xiao, and M. Zhao, "Hasfs: optimizing file system consistency mechanism on nvm-based hybrid storage architecture," *Cluster Computing*, vol. 23, pp. 2501–2515, 2020.

[20] K. Zhou, Y. Zhang, P. Huang, H. Wang, Y. Ji, B. Cheng, and Y. Liu, "Efficient ssd cache for cloud block storage via leveraging block reuse distances," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2496–2509, 2020.

[21] S. Afzal and G. Kavitha, "Load balancing in cloud computing–a hierarchical taxonomical classification," *Journal of Cloud Computing*, vol. 8, no. 1, p. 22, 2019.

[22] "Nvm express moves into the future," 2021. https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf.

[23] "Spdk nvme-of rdma (target & initiator) performance report release 22.09," 2023. https://ci.spdk.io/download/performancereports/SPDK_rdma_mlx_perf_report_2209.pdf.

[24] "Samsung's poseidon v2 e3.x reference system," 2021. https://www.inspursystems.com/product/open-storage/.

[25] "Fio," 2021. https://github.com/axboe/fio.

[26] C. A. Waldspurger, N. Park, A. T. Garthwaite, and I. Ahmad, "Efficient mrc construction with shards.," in *FAST*, vol. 15, pp. 95–110, 2015.

[27] J. Fu, D. Arteaga, and M. Zhao, "Locality-driven mrc construction and cache allocation," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 19–20, 2018.

[28] "Memcached," 2023. https://memcached.org.

[29] "The density, cost, and marketing of semiconductor memory," 2021. https://news.skhynix.com/the-density-cost-and-marketing-of-semiconductor-memory/.

[30] J. Li, Q. Wang, P. P. Lee, and C. Shi, "An in-depth analysis of cloud block storage workloads in large-scale production," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 37–47, IEEE, 2020.

[31] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, pp. 1–23, 2008.

[32] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *Proceedings of the 10th ACM International Systems and Storage Conference*, pp. 1–11, 2017.

[33] "Poseidonos," 2023. https://github.com/poseidonos/poseidonos.

[34] "Block device user guide." https://spdk.io/doc/bdev.html.

[35] The GNOME Project, "GLib – C Utility Library." https://developer.gnome.org/glib/, 2023.

[36] J. Bonwick *et al.*, "The slab allocator: An object-caching kernel memory allocator.," in *USENIX summer*, vol. 16, Boston, MA, USA, 1994.

[37] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, pp. 45–58, 2013.

[38] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou, "S-cave: Effective ssd caching to improve virtual machine storage performance," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pp. 103–112, 2013.

[39] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side ssd caching for storage performance control," in *2015 IEEE International Conference on Autonomic Computing*, pp. 51–60, IEEE, 2015.

[40] J. Fu, Y. Lu, J. Shu, G. Liu, and M. Zhao, "Cowcache: effective flash caching for copy-on-write virtual disks," *Cluster Computing*, vol. 23, pp. 623–639, 2020.

[41] J. Fu, Y. Liu, and G. Liu, "Jcache: Journaling-aware flash caching," *IEEE Access*, vol. 8, pp. 61289–61298, 2020.

[42] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: A Capacity-Optimized SSD cache for primary storage," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 501–512, USENIX Association, June 2014.

[43] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, "vCacheShare: Automated server flash cache space management in a virtualization environment," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 133–144, USENIX Association, June 2014.

[44] G. Yadgar, M. Factor, and A. Schuster, "Karma: Know-it-All replacement for a multilevel cache," in *5th USENIX Conference on File and Storage Technologies (FAST 07)*, (San Jose, CA), USENIX Association, Feb. 2007.

[45] L. Ou, X. He, M. Kosa, and S. Scott, "A unified multiple-level cache for high performance storage systems," in *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 143–150, 2005.

[46] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao, "Second-Tier cache management using write hints," in *4th USENIX Conference on File and Storage Technologies (FAST 05)*, (San Francisco, CA), USENIX Association, Dec. 2005.

[47] A. Amar, A. Raja, and V. Sundararajan, "Glusterfs: a scalable network filesystem," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, USENIX Association, 2004.

[48] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "The design and implementation of a distributed file system," in *Proceedings of the 1985 Summer USENIX Conference*, USENIX Association, 1985.

[49] M. contributors, "MinIO - object storage for the next generation." https://min.io/, 2021. Accessed: March 3, 2023.

[50] Amazon Web Services, "Amazon S3 - simple storage service." https://aws.amazon.com/s3/, 2021. Accessed: March 3, 2023.

[51] Z. Guz, H. Li, A. Shayesteh, and V. Balakrishnan, "Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation," in *Proceedings of the 10th ACM International Systems and Storage Conference*, pp. 1–9, 2017.
[52] M. Nanavati, J. Wires, and A. Warfield, "Decibel: Isolation and sharing in disaggregated rack-scale storage.," in *NSDI*, vol. 17, pp. 17–33, 2017.
[53] S. Sanfilippo, "Redis." https://redis.io/, 2009. Accessed: March 4, 2023.
[54] B. Bulkowski and S. Srinivasan, "Aerospike: Architecture of a real-time operational dbms," *IEEE Data Eng. Bull.*, vol. 36, no. 1, pp. 3–9, 2013.
[55] D. Byrne, N. Onder, and Z. Wang, "Faster slab reassignment in memcached," in *Proceedings of the International Symposium on Memory Systems*, pp. 353–362, 2019.
[56] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter, "RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 195–212, USENIX Association, Oct. 2018.
[57] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "LAMA: Optimized locality-aware memory allocation for key-value cache," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, (Santa Clara, CA), pp. 57–69, USENIX Association, July 2015.
[58] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, (Lombard, IL), pp. 371–384, USENIX Association, Apr. 2013.
[59] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), Association for Computing Machinery, 2014.
[60] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang, "Efficient and available in-memory kv-store with hybrid erasure coding and replication," *ACM Trans. Storage*, vol. 13, sep 2017.
[61] C. Dubnicki and T. J. LeBlanc, "Adjustable block size coherent caches," *SIGARCH Comput. Archit. News*, vol. 20, p. 170–180, apr 1992.
[62] A. J. Smith, "Line (block) size choice for cpu cache memories," *IEEE Transactions on Computers*, vol. C-36, no. 9, pp. 1063–1075, 1987.
[63] S. Przybylski, "The performance impact of block sizes and fetch strategies," *SIGARCH Comput. Archit. News*, vol. 18, p. 160–169, may 1990.
[64] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), p. 454–464, Association for Computing Machinery, 2011.
[65] S. Prybylski, M. Horowitz, and J. Hennessy, "Performance tradeoffs in cache design," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, (Washington, DC, USA), p. 290–298, IEEE Computer Society Press, 1988.
[66] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Trans. Comput. Syst.*, vol. 7, p. 184–215, may 1989.
[67] N. Jeremic, H. Parzyjegla, and G. Muhl, "On adapting the cache block size in ssd caches," in *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–8, 2021.