C2P: Co-operative Caching in Distributed Storage Systems

Shripad J. Nadgowda[1], Ravella C. Sreenivas[2], Sanchit Gupta[3], Neha Gupta[4], and Akshat Verma[1]

[1] IBM Research, India
{nadgowdas,akshatverma}@in.ibm.com
[2] IIT Kharagpur, India
chaitanya.sreenivas@cse.iitkgp.ernet.in
[3] IIT Kanpur, India
sanchitg@iitk.ac.in
[4] IIT Delhi, India
cs1100230@cse.iitd.ac.in

Abstract. Distributed storage systems (e.g. clustered filesystems - HDFS, GPFS and Object Stores - Openstack swift) often partition sequential data across storage systems for performance (*data striping*) or protection (*Erasure-Coding*) . This partitioning leads to logically correlated data being stored on different physical storage devices, which operate autonomously. This un-coordinated operation may lead to inefficient caching, where different devices may cache segments that belong to different working sets. From an application perspective, caching is effective only if all segments needed by it at a given point in time are cached and a single missing segment may lead to high application latency. In this work, we present C2P: a middleware for co-operative caching in distributed storage. *C2P* uses an event-based architecture to co-ordinate caching across the storage devices and ensures that all devices cache correlated segments. We have implemented C2P as a caching middleware for hosted Openstack Swift Object Store. Our experiments show 4-6% improved cache hit and 3-5% reduced disk IO with minimal resource overheads.

1 Introduction

Distributed storage systems often partition sequential data across storage systems for performance (*data striping*) or protection (*Erasure-Coding*) . *Data striping* [5][4] is a technique in which logically sequential data is partitioned into segments and each segment is stored on different physical storage device(HDD). This helps improve aggregate I/O performance by allowing multiple I/O requests to be serviced in parallel from different devices. Striping has been used in practice by storage controllers to manage HDD storage arrays for improved performance for more than a decade (e.g., *RAID 0* [15]) . Most of the popular enterprise cluster/distributed filesystems IBM GPFS[13], EMC Isilon OneFS[14], Luster[17] etc. support data striping. Also, popular blob-storage like Amazon S3[2], Openstack Swift[20], Microsoft Azure[18], Google Cloud Storage[11] support segmented blob uploads.

Logically correlated data in storage systems also gets partitioned due to new data protection techniques like Erasure-Coding (*EC*)[3][12][24], which deliver higher mean time between data loss (MTBDL) as compared to RAID. For example, with 9:3 *EC* data protection policy, when a new data is written, it is first partitioned into 9 equal-sized segments. Next, 3 additional code segments are computed from the data segments. These 12 segments are then stored on different storage nodes. Any 9 of these 12 segments then can be used to satisfy subsequent read requests for the data.This provides availability of the data for maximum up to 3 disk failures. Thus, either for performance or for redundancy, we are increasingly seeing data segmentation in distributed storage systems today.

1.1 Our Contributions

In this work, we specifically study and analyze cache efficiency for distributed storage systems. Typically, the placement policy in these systems is to store each segment on a different storage device/node, which operate autonomously with their own cache management policies. This leads to inefficient caching across all nodes, where different devices may cache segments that belong to different working sets. From an application perspective, caching is effective only if all segments needed by it at a given point in time are cached and a single missing segment may lead to high application latency.

We build the *C2P* system, which implements co-operative caching for distributed storage. *C2P* implements a reliable communication protocol between the cache controllers of individual storage nodes. Through this protocol, each controller communicates *relevant* local cache events (not the data) to the peer nodes. Each node leverages their local cache events and events communicated from peer nodes to implement a *coordinated caching policy*, which ensures that all the logically co-related segments of data will remain in the cache. We have implemented *C2P* for Openstack Swift, which is one of the most popular object stores. Our experiments show that *C2P* improves cache hit for objects by 4-6% and allows 5% of the additional requests to be serve from cache, with minimal resource overheads.

1.2 Paper Overview

The rest of this paper is organized as follows. We provide some background and motivate our problem and solution in Section 2. We also discuss the main challenges we faced, describe the architecture of *C2P*, and discuss certain key design choices. Section 3 describes our implementation of *C2P* and certain optimizations we performed. We evaluate *C2P* and report the results in Section 4. Section 5 discusses related work, and Section 6 highlights the limitations and other potential applications of *C2P*. We finally conclude this paper in Section 7.

2 Design

In this section, we first motivate the need for co-operative caching in distributed storage systems. We discuss few key design challenges for *C2P* and our approach.

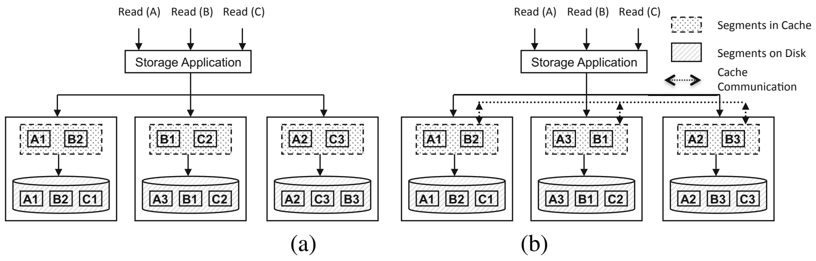


Fig. 1. Distributed Systems with (a) Independent and (b) Co-operative Caches

2.1 Motivation

Let's consider a distributed storage application with 3 storage nodes as shown in Fig.1. Each node has a cache capacity to host only 2 segments. We store 3 objects - *A, B, C* in this storage system. Each object is segmented into *3* partitions and placed on different storage nodes as shown. Also, consider the latency to access a segment from cache to be $50ms$ compared to the *disk latency* of $200ms$. We identify an *object access* as complete only when all its segments are read. Hence, *access latency* is defined as the maximum time taken to read any segment for the object. *Disk IO* is measured as the total segments read from disk across all storage nodes.

Fig.1 (a) shows the cache state at some point in time of a traditional system without any cache co-ordination and (b) shows the cache state of a co-operative co-ordinated cache system. At this stage if objects *A, B and C* are accessed from application, then we can observe the system characteristics as shown in Tab.1. As we can see, for both traditional and *C2P* system total segments hit (6) and miss (3) in the cache are same. Also, number of disk IOs (3) are same. However, applications experience very different access latency with the two systems. In the traditional system without any cache co-ordination, each of the object suffers disk latency (200 ms) in their access. On the other hand, in a co-operative cache system with co-ordination, we are able to reduce the access latency for 2 objects (*A, B*) to cache latency (50 ms) and only 1 object (*C*) incurs disk latency. Hence, if all cache controllers are able to achieve a distributed consensus on the segments to cache, this can lead to improved response time for served objects.

Table 1. Comparison between (a) Independent and (b) Co-operative Caches

	Traditional System				C2P System			
	Cache Hits	cache Miss	Access latency	Disk IO	Cache Hits	Cache Miss	Access latency	Disk IO
read(A)	2	1	200	1	3	0	50	0
read(B)	2	1	200	1	3	0	50	0
read(C)	2	1	200	1	0	3	200	3
Total	6	3	-	3	6	3	-	3

2.2 Design Challenges and Approach

A key appeal of distributed storage systems is their scale and reliability as there are no single points of contention or failure. A co-operative caching system needs to ensure that core distributed nature of the systems is not impacted. Hence, our design space is restricted to peer-based caching, where caching decision on each node is made in a completely distributed manner in contrast to a central controller that makes caching decisions. Each cache controller will implement an identical algorithm and the only change from classical independent cache controller is that each cache controller in *C2P* has access to relevant cache events from all peers. This kind of peer-based co-operative caching poses the following key challenges.

- **Distributed Consensus for Cache Management**: Each node in the *C2P* will have 2 sets of cache information - namely **Local Cache Metadata or LMD** and **Remote Cache Metadata or RMD**. LMD on a node is the repository of the cache events generated by that node while RMD is the repository of the cache events received from the peer nodes. In an ideal situation, all cache controllers need to arrive at a consensus on the objects to be cached in a fully distributed manner. Designing a distributed consensus is challenging and we address this problem by defining global metrics based on the local metrics and remote metrics. Our defined metrics lead to consistent values for objects across all storage nodes in a probabilistic sense. This ensures that even though each cache controller executes a cache eviction policy in isolation, all of them converge to the same object ordering in most cases.
- **Identifying Relevant Events**: Every data operation (READ/WRITE/DELETE) has associated one or more cache event(s). Moreover, same data operation can create different cache events. E.g. READ request for some data might cause <cache miss >or <cache hit >event. It is important to snoop these events very efficiently without adding any overhead to the data path. These captured events then need to be classified into predefined categories. These categories then help implement cache management policies in *C2P* system. E.g. prefetching policy would need <cache miss >category.
- **Peer node discovery**: A set of nodes are identified as **peer** if they are hosting the segments for the same objects. Set of peer nodes is different for each object. Peers are created dynamically and need to be identified quickly to ensure that *relevant* cache events are quickly communicated to peer. We had two design choices here: 1) each node broadcast their events to all nodes but only peer nodes will match and process those events. 2) each node send the events only to its peer nodes. The former option clearly had the downside of overloading the network. Consider, a storage system with 100 nodes where an object with 2 segments is stored will generate 200 events on the network (100 by each node) for each object access. Later option would certainly minimize this overhead. But it has challenge on how a node will discover it's peers for a given object. Storage applications typically decide on the placement of segments for an object dynamically and also stores this mapping. Thus, we could have an *application-tailored* peer node discovery for this purpose. In *C2P* we selected the latter option.
- **Load-proportional Communication Overhead**: Peak load in storage systems are co-related with high number of cache activities (reads, evictions, writes). Hence, more cache activities across nodes generate large number of cache events in the system. As a consequence, the network may become a bottleneck during high load and lead to inefficient caching. We address this problem by implementing an aggregation scheme, which ensures a communication overhead that is almost oblivious to application I/O load. In **aggregation**, cache events are buffered for short duration

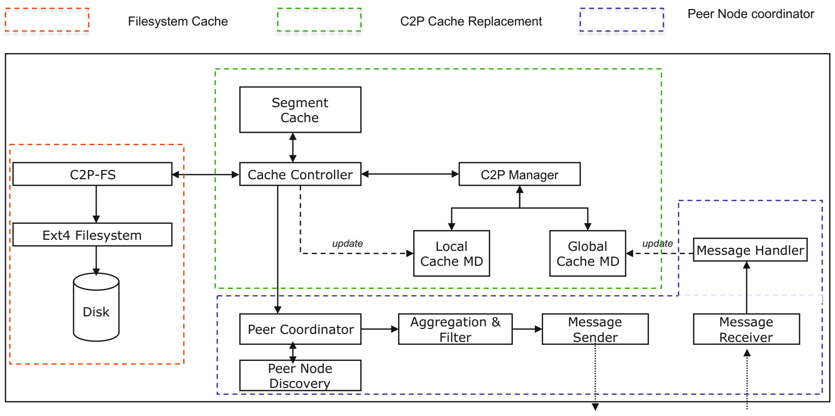


Fig. 2. Architecture of *C2P*

before transmitting and multiple cache events to the same peer node are coalesced together. We also use **filtering** to prioritize and drop low priority events.

3 Implementation

The design for *C2P* in itself can be implemented as an extension to any distributed storage system that supports data segmentation. As a concrete implementation, we have implemented *C2P* into a filesystem cache for open-sourced and widely accepted Openstack Swift - a highly available, distributed, eventually consistent object/blob store. We next discuss the implementation details.

3.1 Filesystem Cache

- **Filesystem:** For implementing *C2P* into a filesystem cache, we decided to use *Filesystem in user space (FUSE)* [10]. FUSE allows us to develop a fully functional filesystem in user's space with simple API library, and it has a proven track record of stability. We call our filesystem implementation C2P-FS. In C2P-FS we have primarily extended read() and write() API calls and other calls are simply redirected to the lower filesystem.
- **Cache:** Similar to "page cache" in traditional filesystem, we have defined "segment cache" in C2P-FS. We have implemented cache using a fixed-size shared memory. Based on size of the workload used during the experiments and heuristics derived from real world scenarios, we configured cache size to be *128 MB* on each storage node. Further, cache line size is changed from page size to segment size i.e. from 4KB to 1 MB. Thus, C2P-FS cache can hold maximum of 128 segments. This change in the cache line size is motivated by three facts: 1) swift application is going to be used for storing/accessing segmented dynamic large objects with segment size of 1 MB 2) partial object access is not available in swift 3) Thus, any object IO (GET/PUT) in swift will cause file IOs (read/write) on C2P-FS on each storage node in the unit of 1 MB. And, having a cache line aligned with the size of IO request is going to boost the performance for any storage system.

Table 2. C2P Data Structures

Event ID	Definition
1	(cache replace) data flushed from the cache
2	(cache add) new data added to cache
3	(cache miss) data is read from disk and added to cache
4	(cache hit) data read from the cache
5	(cache delete) data is deleted from disk and cache

(a) Cache Event Classification

MD field	Definition
path	local filepath of the segment
timestamp	local access time of segment
hitcount	local hit count of segment

(b) Local cache Metadata (LMD)

MD field	Definiition
path	local filepath of the segment
timestamp	global access time of an object
hitcount	global hit count of an object
Object in Cache (OiC)	fraction of object's all segments present in the cache

(c) Global Cache Metadata (GMD)

3.2 Peer Nodes Co-ordination

– **Cache Events:** We first identified and classified the important cache operations that needs to be communicated to the peer nodes as shown in Tab.2(a). For each file IO request in C2P-FS there are going to be one or more cache operations in cache controller. E.g. if the cache is full and there is a read request for data which is not present in the cache, then there will be *cache miss* and *cache replacement* operations in cache controller. For each operation, cache controller then generates a *cache event* and asynchronously sends it for communication. Cache event is a tuple with <event id, file path, timestamp>and size less than 100 bytes. Cache controller also adds this cache event to the *Local Cache MD* (discussed below).

– **Peer nodes discovery:** In object stores, data has different namespaces in storage application (like swift) and filesystem on storage node where it is stored. When user uploads a data, it is identified as "/<container(s)>/<objectid>" by swift. And when it is stored on filesystem is has a filepath like <objectid>/ <timestamp>/ <size>/ <segment>. E.g. for uploading 5 MB data with 1MB segment size user will have command like "swift upload mycontainer myfile" and say on storage nodes node1, node2,...node5 these segments gets stored with filepaths like myfile /1401080357/ 1000000/ 01, myfile/ 1401080357/ 1000000/ 02 ,../05. Since, we have a cache implemented at filesystem, in cache events we can only get filepath for segments. For finding the peer nodes storing the segments for the same objects we made following two changes: 1) inverse lookup for object path from filepath, we changed swift-client to add an extra header "X-Obj-Name:<objpath >" for each segment. This header gets stored as an xattr of the segment file, which we can read to get an object path for a segment. 2) We have developed a new swift service called "swift-discovery" implementing the same protocol as a swift's ring service. Given an object path this discovery service returns a list of peer nodes storing the segments for the same objects and the local filepaths of those segments on the respective storage nodes. For example, calling discovery service on node1 with filepath myfile/../01, will return a peer nodes map as node2:myfile/../02, node3:myfile/../03,....

– **Message Broker:** We use RabbitMQ [22] as a message broker for communicating cache events between storage nodes. We create one message queue per node. Then, for a given cache event and list of peer nodes, it publishes each event in queue of respective peer node. To minimize the network overhead, we implemented

'Aggregation and Filter' policy. In aggregation, before publishing the cache event, we buffer them for short duration of 200 ms. And during this time, if there are more cache events for the same node, then they are aggregated which reduces the payload size. While aggregation is an optimization policy, filtering is a throttling policy. In an overloaded system filtering essentially prioritize and drop some events.

3.3 C2P for Cache Replacement

- **Local cache MD:** Local cache MD (LMD) is the metadata about the segments which are currently present in the segment cache. This MD is maintained in a separate segment of a shared-memory. Each storage node will have their own LMD. The metadata primarily contains 3 fields as shown in Tab.2(b). When a new segment is added to the cache (for cache add/miss) a tuple with <path, current time, 1>is added into this MD. Them for every segment read (cache hit) from the cache this tuple is updated with <current time, hitcount++>. Finally, when a segment is removed from the cache (cache delete/replaced) the MD is removed. In the absence of any co-operation between the storage nodes, cache controller on each node can use this LMD to implement *Least Recently Used (LRU)* replacement policy as described in Algo 1.

Algorithm 1. LRU Comparator

```
1: procedure LRU-COMPARE(candidate a, candidate b)
2:     $affinity_{th} \leftarrow temporal - affinity - threshold$
3:     if $(|a.timestamp - b.timestamp|) < affinity_{th}$ then
4:         if $a.timestmap > b.ttimestamp$ then
5:             return 1
6:         else $a.timestamp < b.timestamp$
7:             return -1
8:         end if
9:     end if
10:    if $a.hitcount > b.hitcount$ then
11:        return 1
12:    else $a.hitcount < b.hitcount$
13:        return -1
14:    end if
15:    return 0
16: end procedure
```

- **Global cache MD:** Global cache MD (GMD) on a given storage node is the metadata about segments hosted on that storage node and which is communicated from the peer storage nodes. Note that, the segments in this GMD not necessarily be in present in the cache but it could also be on the disk as well. Fields of the GMD are shown in Tab.2(c). When an object is read, all it's segments will be accessed generating an cache event for their peer nodes. Also, each node will receive cache event from all peer nodes for a given segment. <timestamp>is the latest timestamp received from cache event for a segment. <hitcount>for a segment is incremented for each cache event received. Thus, when an object is accessed, each node hosting the segments will have *Local hitcount =1* and *Global hitcount = 5*. We normalise *Global hitcount* w.r.t *Local hitcount* to determine an *object size* i.e. number of segments for an object. E.g. For *Global hitcount = 5 and Local hitcount =1* object size = Global hitcount / Local hitcount = 5. Finally, *Object in Cache (OiC)* is the most critical field from the GMD and is being used to implement C2P cache replacement policy. OiC is defined as the fraction of all segments of an object available in cache across all peer nodes. Considering that with swift, there won't be any partial object access, *OiC* is reset to 1 for any *cache add or cache miss* event. And it is recomputed as follows: first $segeval$ is computed as (1/object size) which is the fractional value for each segment of an object. Then, for every *cache replace* event OiC is decremented as (OiC = OiC - segeval). Fig.3 shows along a timeline, few sample cache events and how those are reflected into GMD state on one of the storage node for one segment

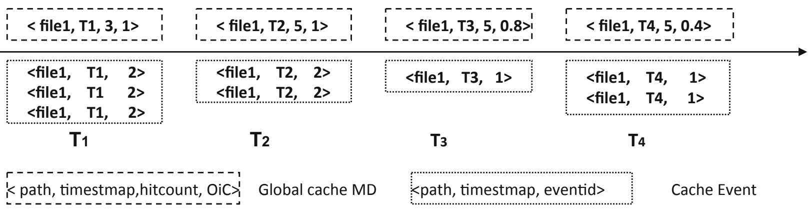


Fig. 3. GMD State Transition

- **Hitcount Decay:** We have defined exponentially decay function on hitcount of an object. If an object is not accessed recently (not within last 60 secs, which is a heuristically derived period) then, we decrement the current hitcount by factor of 0.8. This decay ensures cache-fairness through normalization of hitcount for objects which are popular for short period (gets high hitcount in short time).
- **C2P Cache Replacement:** To demonstrate the effectiveness of co-ordinated caching in distributed storage systems, we have designed and implemented a cache replacement policy on C2P system named *C2P-CR*. We have also implemented the LRU replacement policy to simulate the one in traditional systems. In LRU we sort all the candidate cache MDs using an LRU-Comparator function as described in Algo. 1. In this function, we first measure the *temporal affinity* between the two candidates, which is essentially difference between their timestamps. If the affinity is less than a (heuristically derived) threshold, then they are sorted based on their timestamps. Otherwise, they are sorted from their hitcount. For *C2P* we have defined a C2P-Comparator for sorting candidates to be selected for cache replacement as described in Algo.2. In this we leverage both LMD as well as GMD to select the segment(s) to be replaced from cache. First *Object in Cache*(OiC) is computed from GMD as discussed earlier and candidates are compared based on OiC. If the candidates have same OiC, then those candidates are sorted using LRU-Comparator. Thus, *C2P-CR* is essentially built on top of LRU.

Algorithm 2. C2P CR Comparator

```
1: procedure C2P-COMPARE(candidate a, candidate b)
2:     if $a.OiC > b.OiC$ then
3:         return 1
4:     else $a.OiC < b.OiC$
5:         return -1
6:     end if
7:     return LRU-Compare(a,b)
8: end procedure
```

4 Evaluation

We evaluate *C2P* with Openstack Swift Object store. Swift was deployed on a set of 8 VMs running Ubuntu 12.04 LTS. The VMs were hosted on 2 servers each with 24-core 3:07GHz Xeon(R) processor and 64GB memory. Each VM was configured with 2 vC-PUs, 2 GB memory and a 50 GB disk formatted with ext4 filesystem. We configured 128 MB cache size for C2P-FS on each VM. This cache size was decided based on heuristics and size of our workloads. We have defined two configurable modes of cache management for C2P-FS - namely C:ON and C:OFF. C:ON indicates that co-operative caching policy is ON for cache replacement on all storage nodes while C:OFF indicates that each node implements default *LRU* cache replacement policy. We evaluate *C2P* based on several metrics. First, in the *baseline experiments*, we measured the overhead of our cache implementation by comparing the performance with native implementation of fuse. Then, in the *case study* experiment we specifically measured the cache efficiency with *C2P* cache replacement policy against traditional *LRU: Least recently used*.

We tag all the data access (read) on each of the individual storage node either as a *segment hit* or a *segment miss*. *segment hit* indicates that the data is read from the cache while the later indicates that data is read from the disk. More importantly we also tag each object access. When each segment of an object is a *segment hit* we identify it as an *Object hit*. If there is a *segment miss* for even a single segment, it is an *Object miss*. We further decompose *Object miss* into *Object miss complete* and *Object miss partial* to indicate whether there is a *segment miss* for all the segments or some segments respectively . We define *comm latency* as the delay between the times when a cache event is published by any storage node and when it is delivered to peer nodes. We also measured *comm overhead* as the number of messages (cache events) generated per second. Finally, we measured *object throughput* as size of object (in MB) read per second.

4.1 Baseline Experiment

In this section, we discuss baseline experiments that we conducted to evaluate the performance overhead of our cache implementation with fuse [10] filesystem. We conducted these experiments in two phases. In the first phase, we used standard swift deployment wherein each storage node had ext4 filesystem with it's native *LRU* cache replacement policy. In the second phase, we deployed swift with our C2P-FS in C:OFF mode i.e.cache

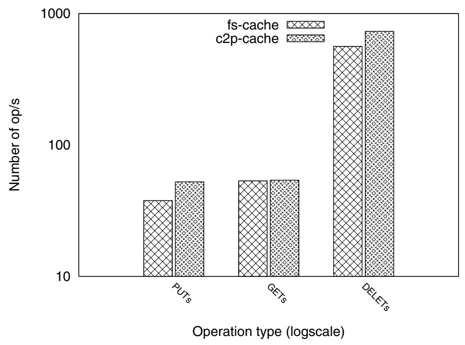


Fig. 4. Swiftbench Evaluation

co-operation is disabled and default *LRU* cache replacement policy is used on storage nodes to match standard setup. We used *Swift-bench* [21] which is a standard benchmarking tool for Openstack Swift. We chose three common IO workloads on any object store - namely PUT, GET and DELETE for these experiments. We further define the workload profile with 500 PUTs, 2000 (random) GETs and 500 DELETEs for object size of *1 MB*. Then, we ran this same workload profile in both phases and measured operation throughput as shown in Fig.4. As we can see, C2P-FS achieves almost the same throughput as with the standard filesystem deployment for all three kinds of workloads. Thus through these baseline experiment we established that our C2P-FS cache implementation does not incur any performance overhead over a standard swift deployment. Hence, in the case study experiments below we used C2P-FS in C:OFF mode as a reference system implementing *LRU*. Then we compared and contrasted the metric measurements of *C2P* system against it.

4.2 Case Study

In case study experiment, we try to motivate application of *C2P* for distributed storage system hosting a segmented or striped data for improved cache efficiency.

Data Store: We first uploaded 500 objects of size *5 MB* each. During upload, we split the object with *1 MB* segment size using swift's support for dynamic large object. Ideally, we would expect each segment to be stored on different storage node. But, swift uses a ring service for placement which does not guarantee this segment isolation for a given object. We captured segment placements for all the objects across 8 storage nodes in a heatmap shown in Fig.5. As we can see for some objects maximum of 4 segments are stored on a same storage node. We also measured total number of segments stored on each storage node. Fig.6(b) shows the distribution of all (500 x 5 = 2500) segments across 8 storage nodes. The distribution is not even across the nodes, and this is typically true for all the distributed systems.

Access pattern: We used powerlaw to generate a long tail distribution series with 2000 numbers in the range [0:500] which would mimic a real-world application access pattern. Such series typically identifies a workset which contains few popular objects that are accessed more frequently as shown in Fig.6(a). We numbered all objects from the data store from 1-500 and then used this series to identify the object number to

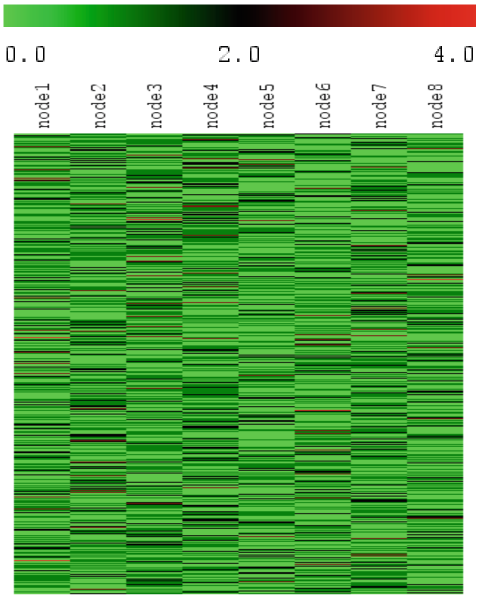
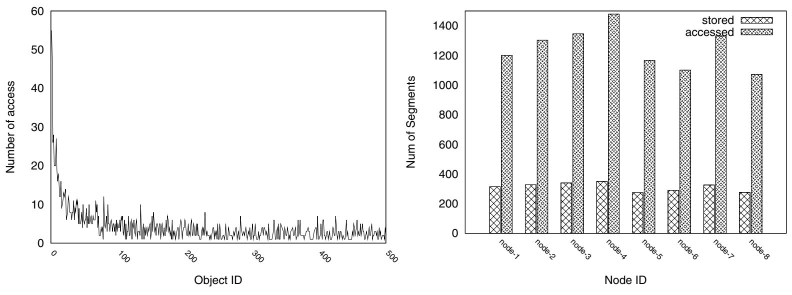


Fig. 5. Segment Placement heatmap for objects

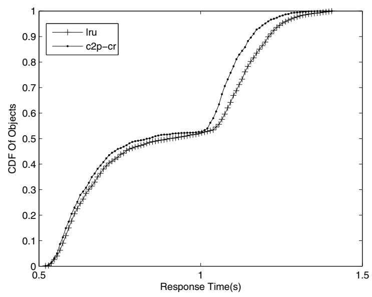
access. When we access an object, we access it completely i.e. all 5 segments. But, even for partial access *C2P* efficiency will be the same. We also measured total segments accessed across all storage nodes as shown in Fig.6(b). As we can see, there is a large variation in the access load across storage nodes which is again mostly true for all distributed storage systems.



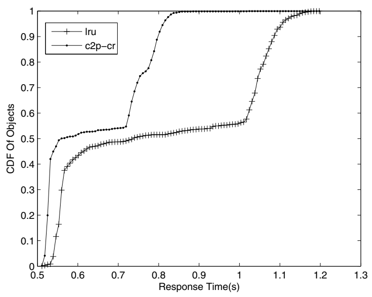
(a) Object access frequency (b)Segment/Access Distribution

Fig. 6. Data Store and Access pattern

Thus, this un-even segment distribution compounded with variant access load creates an erratic data pressure across storage nodes in distributed storage systems. Thus, there is a greater need of co-operation to enable highly utilized nodes to mantain their cache states in consistent with their peer nodes which are less utilized.



(a) Multithread

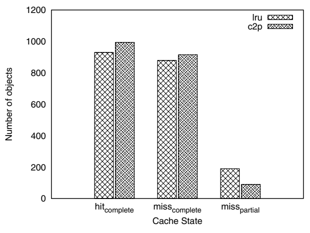


(b) Singlethread

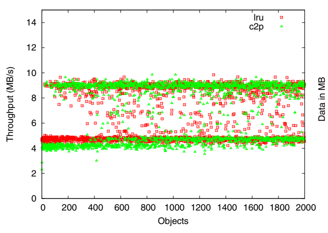
Fig. 7. Response Time CDF

Single-threaded Run. In single-threaded run we used a single swift client which would read objects from the data store following an access pattern. In the results, first we analyze the most important aspect from application's point of view i.e. *Object hit* ratio. As shown in Fig.8(a) for C2P systems we get almost 6.7% more *Object hit* in the cache. Amongst object misses, we measured around 50% reduction for *Object miss partial* and 4% increase for *Object miss complete*. Putting these numbers in perspective, we note that for applications storing segmented objects, C2P system can help achieve better cache efficiency at object level to reduce an application latency. In Fig.7(b), we also plot cdf of number of objects against their response time. This is an important measure which can be translated into SLA assurance of a storage system. For example, for SLA of *response time <0.8 sec C2P* system has about 6% more objects satisfying the SLA than the one implementing *LRU*. Another interesting observation we made here is that, for cache missed objects response time for *C2P* is between $0.7s$ to $0.9s$ while that for LRU is between $0.7s$ to $1.2s$. We conjecture that this increased latency for LRU for cache missed objects is attributed to the increased disk queue length for missed segments. Fig.8(b) shows the *object throughput* measured for each object access. It shows for *C2P*, most of the objects have either high throughput around *9 MBps* (Object hit) or low throughput around around *4 MBps*(Object miss complete). While for LRU there are many objects with throughput in between (Object miss partial). As mentioned earlier, for *C2P* we increases *Object miss complete*, but that does not necessarily means disk IO is also increased. To elaborate on this, for each object accessed we also traced *segment hit* ratio on each storage node. As shown in Fig.8(c) on each individual storage node we get more *segment hits*. In effect, we reduces disk IO on each storage node and overall we observed about 5% reduced disk IO across all storage nodes which is a very critical measure for any storage system. Finally, Fig.10 shows the Rabbit MQ's monitored state of the message queue. As we can see, C2P system requires around 20 messages/s to cache event co-ordination and *comm latency* less than 200 ms. And considering size of each message is less than 100 Bytes, the network overhead is very minimal.

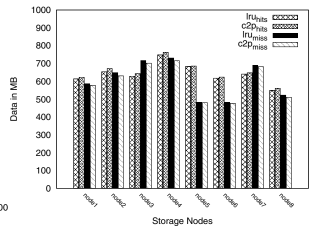
Multi-threaded Run. In multi-threaded run we used 4 swift clients. We split the access pattern of 2000 objects into 4 access patterns of 500. Then, we ran all the 4 clients in parallel requesting objects from the respective split-access pattern. Similar to Single-threaded run, we measured system characteristics across different metrics. Fig.9(a) shows 4.5% improved *Object hit*, and amongst object misses 43% reduced *Object miss partial* and around 4% increases for *Object miss complete*. Fig.7(a) shows the cdf of number of objects against their response time. Again, compared to *LRU* in *C2P* we measured larger % of objects under any given response time. Fig.9(b) shows *object throughput* for object access across all 4 clients. We observe similar pattern to that of a singlethread run. And Fig.9(c) shows *segment hit* distribution across all storage nodes. Again, on each storage node we observe better cache hits for C2P, thus reducing the disk IO by around 3.5% across all nodes. Fig.11 shows *comm overhead* in the order of 70 messages/s (7KBps) still very minimal. But, now we get *comm latency* in around 1 second. *Comm overhead* and *latency* are observed to be higher than the respective numbers in the single-threaded run. This is because in multithreaded run, object request rate is higher being coming from 4 clients in parallel which in turn increases the rate of cache activities on individual storage nodes, thus cache events are published at a higher rate. We observed, *Object hit* for C2P system in this run is slightly less than that in the single threaded run. This is attributed to the higher *comm latency* which increases the delay between cache co-ordination across storage nodes. In out future work, we will try to minimize this effect of latency on cache efficiency.



(a) Object Cache Hit/Miss

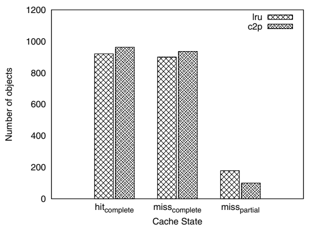


(b)Object Throughput

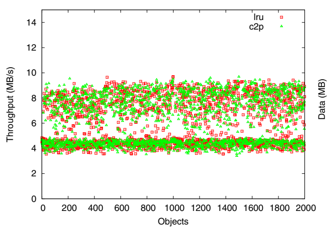


(c) Cache Stats on storage nodes

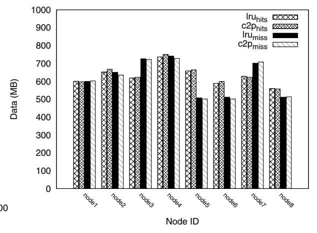
Fig. 8. Single Threaded Data Access Evaluation



(a) Object Cache Hit/Miss



(b)Object Throughput



(c) Cache Stats on storage nodes

Fig. 9. Multi Threaded Data Access Evaluation

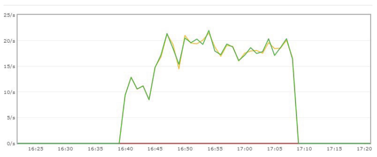


Fig. 10. Singlethread run overhead

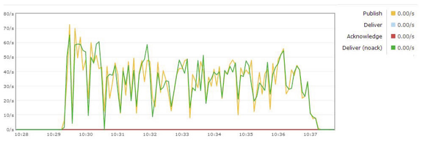


Fig. 11. Multithread run overhead

The maximum value of *comm latency* for optimal performance of C2P system, is a function of size of cache on each storage node. Although, it is important to note here that effectiveness of C2P systems is not limited by *comm latency* of less than a second. But, since we had a small cache size on storage nodes,

To summarize the case study results we note: 1) compare to traditional *LRU* cache replacement policy, C2P achieves 4-6% increase in the object hits thus reducing the access latency for more objects. 2) In C2P systems, on each of the comprising storage nodes cache hits are improved reducing the disk IO by around 3-5%. And 3) event-based architecture to co-ordinate caching incurs a very minimal network overhead.

5 Related Work

Distributed systems and cache coordination techniques in such systems has been around for a long time[1][23][7][6][16]. But cache cooperation traditionally been applied in contexts like scaling or disk IO optimization. To our best knowledge *C2P* is the first system designed to maximize cache efficiency of distributed storage hosting segmented data.

Scale: Memcached[9] is a general-purpose distributed memory caching system. For a large scale cache requirements, hundreds of nodes are setup. And these nodes then leverages their memory through memcached to build a large in-memory key-value store for small chunks of arbitrary data. Facebook probably is the world's largest user of memcached[19].

Disk IO optimization: CCM[8] probably is closet to our work. For cluster-based servers, CCM keeps an accounting information for multiple copies of the same data (blocks) available in the cache across all nodes. Then, this information is used to forward IO requests between nodes to ensure cache hit. Essentially, they increase network communication to reduce disk access. Similarly in [1], technique of split caching is used to avoid disk reads by using the combined memory of the clients as a cooperative cache. DSC[16] describes the problems of the exposition of one nodes resources to others. As they state, cache state interactions and the adoption of a common scheme for cache management policies are two primary reasons behind the problem. [6] mentions interesting techniques for data prefetching with Co-Operative Caching. The bottomline in all these prior work is that - a cache cooperation will happen between the nodes only if they contains the same data.

In *C2P*, the primary distinction is that cache cooperation is designed for *logically related* data e.g. different segments of the same object. Also, there is no resource exposition between the nodes in the cluster i.e. each node will serve the ONLY IO requests for which it is actually hosting the data. Thus, IO requests are not forwarded between the nodes, but just the cache events are communicated.

6 Limitations and Future Work

C2P design presented in this paper caters to the distributed systems storing segmented data and ensures better cache efficiency for them. In our current implementation we have exploited this cache cooperation only for *cache replacement* policy. We also plan to implement and exercise *cache prefetching* for *C2P*, wherein we can prefetch segments based on cache events from the peer nodes. We believe such prefetching will further improve the cache efficiency.

One of the data property we haven't considered in *C2P* is - *replication*. For storage systems, data striping and replication can be applied simulteneously. Here, first we need to understand placement and access characteristics of such data. Then for these scenarios, through cache cooperation we can ensure only one copy of the data segment remains in the cache across all nodes. And these segments in cache might belong to different replica copy of the data.

Finally, we plan to deploy *C2P* in some production distributed system and measure the scalability, overhead for live data.

7 Conclusion

In this paper we present *C2P*: a coperative caching policy for distributed stoarge systems. *C2P* implements a coordination protocol wherein each node communicates their local cache events to peers. Then based on these additional cache state information of peers, each node implements a co-ordinated caching policy for *cache replacement* and *cache prefetching*. These policies in turn ensures a consistent caching across nodes for segments of the data which are logically related. Thus, we can reduce the access latency for the data and improve the overall performance of the system.

References

1. Adya, A., Castro, M., Liskov, B., Maheshwari, U., Shrira, L.: Fragment reconstruction: Providing global cache coherence in a transactional storage system. In: Proceedings of the 17th International Conference on Distributed Computing Systems, pp. 2–11. IEEE (1997)
2. Amazon: Amazon S3, `http://aws.amazon.com/s3/`
3. Bloemer, J., Kalfane, M., Karp, R., Karpinski, M., Luby, M., Zuckerman, D.: An xor-based erasure-resilient coding scheme (1995)
4. Cabrera, L.F., Long, D.: Using data striping in a local area network (1992)
5. Cabrera, L.F., Long, D.D.E.: Swift: Using distributed disk striping to provide high i/o data rates. Computing Systems 4(4), 405–436 (1991)
6. Chi, C.H., Lau, S.: Data prefetching with co-operative caching. In: 5th International Conference on High Performance Computing, HIPC 1998, pp. 25–32. IEEE (1998)
7. Clarke, K.J., Gittins, R., McPolin, S., Rang, A.: Distributed storage cache coherency system and method, US Patent 7,017,012 (March 21, 2006)

8. Cuenca-Acuna, F.M., Nguyen, T.D.: Cooperative caching middleware for cluster-based servers. In: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing, pp. 303–314. IEEE (2001)
9. Fitzpatrick, B.: Distributed caching with memcached. Linux Journal 2004(124), 5 (2004)
10. Fuse: Filesystem in Userspace, `http://fuse.sourceforge.net/`
11. Google: Google Cloud Storage, `http://cloud.google.com/Storage`
12. Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., Yekhanin, S., et al.: Erasure coding in windows azure storage. In: USENIX ATC, vol. 12 (2012)
13. IBM: Introduction to GPFS 3.5 - IBM. RedBook (2012)
14. Isilon, E.: EMC Isilon OneFS: A Technical Overview. White paper (2013)
15. LACIE: RAID Technology. White paper
16. Laoutaris, N., Smaragdakis, G., Bestavros, A., Matta, I., Stavrakakis, I.: Distributed selfish caching. IEEE Transactions on Parallel and Distributed Systems 18(10), 1361–1376 (2007)
17. Luster: Lustre Filesystem, `http://wiki.lustre.org`
18. Microsoft: Microsoft Azure, `http://azure.microsoft.com`
19. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P.: et al.: Scaling memcache at facebook. In: Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation, pp. 385–398. USENIX Association (2013)
20. Openstack: Openstack Swift, `http://swift.openstack.org`
21. Openstack: Swiftbench. https://launchpad.net/swift-bench
22. RabbitMQ: RabbitMQ., `https://www.rabbitmq.com/`
23. Sarkar, P., Hartman, J.H.: Hint-based cooperative caching. ACM Transactions on Computer Systems (TOCS) 18(4), 387–419 (2000)
24. Weatherspoon, H., Kubiatowicz, J.D.: Erasure coding vs. replication: A quantitative comparison. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 328–337. Springer, Heidelberg (2002)