Coral: A Cloud-Backed Frugal File System

Cheng Chang, *Student Member, IEEE*, Jianhua Sun, and Hao Chen, *Member, IEEE*

Abstract—With simple access interfaces and flexible billing models, cloud storage has become an attractive solution to simplify the storage management for both enterprises and individual users. However, traditional file systems with extensive optimizations for local disk-based storage backend can not fully exploit the inherent features of the cloud to obtain desirable performance. In this paper, we present the design, implementation, and evaluation of Coral, a cloud based file system that strikes a balance between performance and monetary cost. Unlike previous studies that treat cloud storage as just a normal backend of existing networked file systems, Coral is designed to address several key issues in optimizing cloud-based file systems such as the data layout, block management, and billing model. With carefully designed data structures and algorithms, such as identifying semantically correlated data blocks, kd-tree based caching policy with self-adaptive thrashing prevention, effective data layout, and optimal garbage collection, Coral achieves good performance and cost savings under various workloads as demonstrated by extensive evaluations.

Index Terms—Cloud storage, file systems, cost optimization, cache, Billing model

1 Introduction

The platform-as-a-service (PaaS) cloud storage has become an infrastructure service of the Internet as a promising way to simplify storage management for enterprises and individual users. Coupled with the increasing demand for multi-device data synchronization and sharing, it is emerging as a new paradigm that helps migrate storage applications to the cloud. Due to its practical impact, significant research endeavors have been undertaken to address the problems in cloud storage based applications, such as the security of storage outsourcing, data consistency, and cost optimization.

A large body of work has advanced the state of art of cloud storage research, including but not limited to the topics mentioned above. In particular, a recent work [38] proposed a cloud-based storage solution called Bluesky for the enterprise, which acts as a proxy to provide the illusion of a traditional file server and transfer the requests to the cloud via a simple HTTP-based interface. By intelligently organizing storage objects in a local cache, Bluesky serves write requests in batches using a log-structured data store and merges read requests using the range request feature in the HTTP protocol. In this way, many accesses to the remote storage can be absorbed by the local cache, avoiding undue resource consumption accordingly. As for cost optimization, FCFS [35] proposed a frugal storage model optimized for scenarios concerning multiple cloud storage services. Similar to local hierarchical storage systems, FCFS integrates cloud services with very different price structures. By dynamically adapting the storage volume sizes of each service, FCFS reduces the cost of operating a file system in the cloud.

In this paper, we present the design and implementation of a cost-effective file system based on the PaaS cloud storage. In contrast to current research activities that view cloud service as a backend for network file systems and adopt classical caching strategies and data block management mechanisms, we argue that many inherent characteristics of cloud storage, especially the billing model, should be considered in order to improve resource utilization and minimize monetary cost. Specifically, we exploit the semantic correlation of data blocks, and propose a system design that takes advantage of a smart local cache with effective eviction policy and an efficient data layout approach. Evaluations performed on our proof-of-concept implementation demonstrate prominent savings in cost and gains in performance as compared to the state of the art.

The main challenge in designing Coral is how to manage data blocks effectively. Comparing to the potentially unlimited storage capacity in the cloud, the relatively small cache on the client side may result in severe performance degradation due to the low cache hit rate [26] and the high latency of the WAN. Thus, a better cache design is needed in order to save the operational cost for a cloud based file system. To this end, we propose a cache eviction mechanism based on kd-tree [19] that is organized by considering the correlation metrics among data blocks. The proposed mechanism guarantees high performance in identifying and evicting cached content with flexible range selection of data blocks. Furthermore, the data layout of concrete objects in the cloud also reflects the semantic correlation of data blocks, which has direct and important implications for several optimizations in our system, including data prefetching, cache thrashing minimization, and garbage collection. Integrated with the quantified model from analyzing the billing items of the cloud service, Coral orchestrates the interfaces provided by the cloud vendor and achieves improvement in both execution time and monetary cost as compared to existing

- *C. Chang is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan, China. E-mail: chengchang@aimlab.org.*
- *J. Sun and H. Chen are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, Hunan, China. E-mail: {jhsun, haochen}@aimlab.org.*

Manuscript received 4 Sept. 2014; revised 9 Feb. 2015; accepted 9 Apr. 2015. Date of publication 19 Apr. 2015; date of current version 16 Mar. 2016. Recommended for acceptance by G. Tan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2424705

TABLE 1
Price Structure of Common PaaS Cloud Storage

Billing Items		Amazon	Rackspace	Google
Storage ($/GB/Month)		0.095	0.100	0.085
	Up	0	0	0
Transfer ($/GB)	Down	0.120	0.120	0.120
	Up	5×10^{-6}	0	10^{-5}
Request ($/Req.)	Down	4×10^{-7}	0	10^{-6}

systems. For example, Coral can save the monetary cost by 28 percent averagely and achieve latency improvement by up to 83 percent.

We make the following contributions.

- We extensively investigate the optimization of monetary cost for PaaS cloud storage with specific billing models, and consequently design optimal strategies for data layout and garbage cleaning to avoid performance degradation in WAN environments.
- We characterize the data block relationship using high level semantics, and then use this information to determine the composition and layout of storage objects. Together with the kd-tree, we implement an efficient cache management component with an adaptive mechanism to alleviate cache thrashing.
- We implement a prototype system called Coral to demonstrate the effectiveness of our proposed solutions for cloud-based file systems. By comparing with an representative log-structure based design, we show that Coral achieves our design goals through experiments with both simulation and real scenario benchmarks.

2 Background and Motivation

This section presents an overview about the cloud storage and motivates the design choices behind Coral. We analyze the differences between the cloud and local environment on data block management. Using an illustrative example, we highlight the design principles in our work.

2.1 Cloud Storage Model

In the present marketplace, cloud storage is a competitive field that provides rich choices over the cloud stack, i.e. infrastructure as a service (IaaS), PaaS, and software as a service (SaaS). We focus on the PaaS cloud storage after considering the data management granularity. In particular, typical PaaS storage services exports simple interfaces such as GET, PUT, LIST, and DELETE to manipulate data objects, and the incurred costs due to these operations are tied to different billing items. Table 1 lists the price of several major cloud vendors [2], [8], [13].

In terms of data block management in file systems, existing solutions enable data to automatically move around in the hierarchical storage stack, which incorporates fixed-size storage medium with distinct performance/price ratio to reduce data storage cost. In addition, an extensive body of research on cache eviction policies, such as the least recently used (LRU), coordinates the action of data blocks in order to minimize overall cache miss rate. As the main issues in block management, the caching algorithm and data organization underlie the fundamental principles in designing storage systems. However, mainstream file systems are designed for single devices or local area network (LAN) environments. We argue that current block management methods are not cost-effective for the cloud. Imagine that if we simply adopt the conventional block management approaches in the cloud, the fees charged for bandwidth consumption and data transfer may be unacceptable for a moderate-sized system, and most importantly, the prohibitively high latency may render the system unusable.

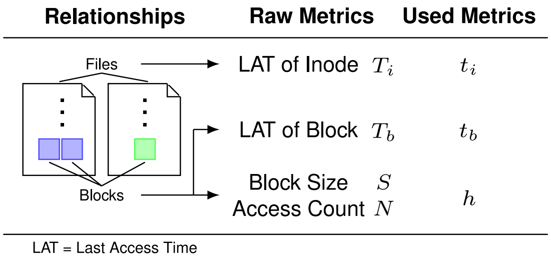


Fig. 1. Metrics and notations.

2.2 Zooming in the Relationship of Data Blocks

Considering the inherent features of the cloud and the runtime dynamics exhibited by data block accesses, we revisit the relationship reflected by this dynamism from the following three aspects. First, some files tend to be operated with the same pattern such as source code files in a project (editing, debugging, and compiling), and consequently the blocks belonging to different files may have a hidden linkage. Second, by analyzing the access mode of blocks in the same file, we can identify whether there exists a group of blocks in a file with distinctive operations associated. Third, similar to the existing temporal-locality based caching policies that strive to discover the most-likely-to-be-reused data, the hotspots identified by analyzing the access patterns should be leveraged to pinpoint the most performance-critical data blocks.

To observe how the data block relationship are exhibited in a real experiment that spans a relatively long period of time, we ran a benchmark of compiling the Linux kernel that incurs massive file read and write operations in the file system. Our experiment simulates a functional cache with fixed size, which evicts data blocks using the LRU policy and loads data blocks to the cache when cache miss occurs. Moreover, we record and calculate the variables that reflect the aforementioned relationship for each block at eviction time. We first analyze the benchmark results and then illustrate the selection of metrics.

Fig. 2 shows the diagram that represents the status of all cached blocks at a specific time instant in 3d coordinates. The notations are illustrated in Fig. 1, where raw metrics (access time, count, and block size) are transformed and normalized by mapping onto the interval [0, 1] as explained later. The recorded information combined with the logs exported by the cloud vendor makes it possible to derive accurate block relationships and corresponding operations. Three typical patterns are explained as follows. First, the compiler creates temporary files such as *.module_name.o.tmp* for each module, which are then transformed to files such

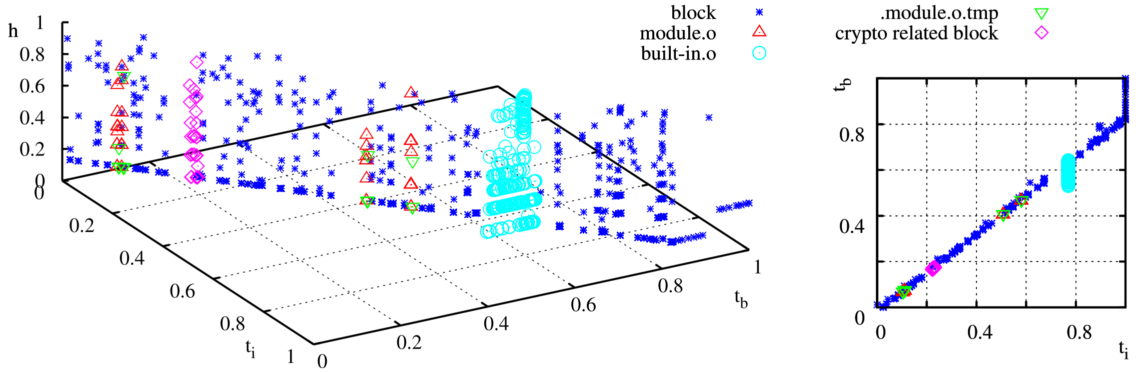


Fig. 2. Block relationship illustration.

as *module_name.o* after compilation optimizations. Therefore, the two types of files indicate a strong linkage. In the diagram, the triangles represent the blocks for three modules and their temporary files (*kprobes.o*, *inode.o*, *aes_generic.o*). Second, the diamond shaped blocks comprise three different modules (*chainv.o*, *crypto_hash.o*, *eseqiv.o*), which are compiled perceivably at the same time. According to the structure of Linux kernel source code, they belong to the cryptography framework (under the *crypto* directory), which is a relatively small subsystem. Hence, they are manipulated within a short time window. Third, it can be observed that the file *built-in.o* marked as circles is accessed at one time (see the t_i dimension), while its blocks exhibit distinct behaviors in access time and hotness (see the t_b dimension, it varies from 0.5 to 0.7). By analyzing the makefile, we found that during the compilation each subsystem generates a *built-in.o* file by linking all individual object files. This example illustrates that it may also be profitable to utilize the relatedness of blocks within a single file

$$
\begin{aligned}
t_i &= \frac{T_i - T_0}{\Delta t}, \\
t_b &= \frac{T_b - T_0}{\Delta t}, \\
h &= \frac{2}{\pi} \cdot \tan^{-1} \frac{N}{S}.
\end{aligned}
\tag{1}
$$

In cache management, block is the basic operational unit, and each block belongs to a specific file pointed to by an inode. Therefore, the last access time for inode (T_i) and block (T_b) is selected to describe the correlation of blocks according to the analysis in the above example, because the closer the access time, the higher relevance the blocks may have. To mask the difference of access frequency among diverse applications (e.g. benchmark applications typically issue frequent requests during short time windows, but normal applications exhibit less such behaviors), we use relative time to represent T_i and T_b as shown in Equation (1), where Δt is the time interval measured between two events of cache eviction. Starting from the latest eviction at time T_0, we collect the statistics about the last access time for each block and file in the cache, and when a new eviction event is triggered, the relative time can be calculated based on Equation (1). The third dimension is about hotness of data blocks, which can be characterized by block access count N and block size S. In order to avoid the well-recognized cache pollution problem due to workloads exhibiting weak locality [23], we use N/S to represent block hotness. This metric is normalized using the nonlinear function $y = 2\tan^{-1}x/\pi$, which has the feature of enlarging smaller x and making y approach 1 when x increases. In this way, smaller x values should be more distinguishable when they are used to describe the hotness metric. For example, for a specific block (the block size s is fixed), the degree of difference between small N values (i.e. 3 and 10) would be more significant than between large ones (i.e. 3,000 and 10,000), because large N values will always fall into the category of hot data (approaching 1) after the nonlinear transform.

2.3 Motivation

In fact, Fig. 2 is just a representative frame extracted from an animation rendered by casting the traced data into a 3D space during offline analysis. Moreover, we experimented with a variety of workloads provided by the file system benchmark tool Filebench [5], and observed invariable features as indicated in the sampled frame above, by manually examining the key frames in dynamically changing scenes.

Based on the above observations and analysis, we propose to consider the following design principals when developing data block management frameworks optimized for the cloud.

- Caching strategy should be re-evaluated. The cloud storage is best characterized by its elastic capacity. As the data volume increases and eventually far exceeds the cache size, the cache hit rate might be significantly reduced when using conventional locality-based caching policies, such as LRU.
- Data layout of storage objects should be reconsidered to take into account the access interface and pricing model of the cloud. The cloud storage service masks the diversity of needs by presenting a key-value store abstraction to the user. Further, the billing model provided by current cloud vendors is typically based on storage capacity, access frequency, and data transfer size. Consequently, an optimal data layout would be beneficial to applications and end users in terms of performance and cost respectively.

3 Design and Implementation

We start with analyzing the impact of the billing model on design choices, and then present an overview of the system architecture. Next, two core components, namely the caching policy and data layout are discussed before some implementation issues in constructing each component are presented.

3.1 Analyzing the Billing Model

We first discuss what comprises the cost of storing an object in the cloud, and then analyze how each individual operation affects the overall cost.

3.1.1 Cost in the Object Life Cycle

For each object stored in the cloud, its life cycle starts from the PUT operation and ends with the DELETE operation. During this time period, the client performs GET operations to retrieve the object multiple times as desired. Since there is no an equivalent UPDATE operation, we have to upload a new object and delete the old one when the content of an object is partially modified. This also implies the beginning of the life cycle for a new object. Additionally, we use a built-in *Database* that acts as the information manager instead of using the cloud storage metadata operation such as LIST to query existing key-value pairs. Thus, the main cost C for each object in its life cycle can be formulated as:

$$C = mR_d + R_u + u(t - t_0)S + umT_d + uT_u, \qquad (2)$$

where R_u/R_d, T_u/T_d is the bidirectional (up/down) unit price for the request and data transfer respectively, and S is the storage price. These uppercase parameters are provided by cloud vendors. We therefore concentrate on how to reduce the number of requests m, storage volume size u, and storage duration $\Delta t = t - t_0$ in order to minimize C.

3.1.2 Effects of Operations on Monetary Cost

Despite the fact that the storage system may perform multifaceted operations on data, we only focus on optimizing the parameters mentioned above and analyzing the indicated operational cost. The operations we consider include caching, split, merge, and compression, which are derived from the requirements of the caching policy and data layout, and consequently have direct relation to the overall performance and significant implication on monetary cost.

Caching and prefetching. According to Equation (2), both effective caching and pre-fetching can reduce the number of request m, indicating decreased cost of $(R_d + uT_d)$ upon each attempt to retrieve the object. Improving local cache hit rate, on the other hand, can also improve response latency. If objects to be accessed in the near future can be accurately predicted, with pre-fetching we can download them in advance to improve the cache hit ratio and save monetary cost accordingly. On the premise of identifying correlation metrics, it is helpful to allocate more computing resource to collect the access patterns for improving the cache hit rate and adapting to the bottleneck WAN link. As the key design point, we elaborate the specifics in Section 3.4.

Split. As exemplified in Section 2.2, splitting the file and storing each portion separately will increase the cost. However, in certain scenarios such as multi-threaded downloading, it would be beneficial to do so to accelerate the fetching of object from the cloud, albeit at the expense of incurring more GET requests. Assume that an object is split into p portions, then the new cost can be formulated as:

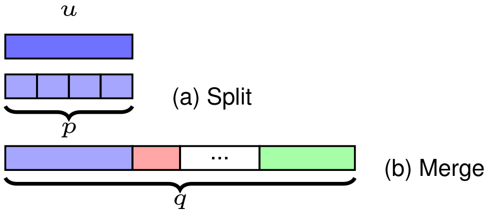


Fig. 3. Splitting and merging example.

$$C_{split} = pmR_d + pR_u + u\Delta tS + umT_d + uT_u. \qquad (3)$$

For an object of size u, the difference between C and C_{split} is given in Equation (4), while Fig. 3a illustrates the splitting process. We can see that when $p = 1$ (namely no split), the cost remains unchanged, and if $p > 1$, each portion raises the cost by $(mR_d + R_u)$

$$C_{split} - C = \begin{cases} (mR_d + R_u)(p-1) & , p > 1, \\ 0 & , p = 1. \end{cases} \qquad (4)$$

Since the split operation leads to extra cost, our system restrictedly employs it in metadata management when a single file exceeds certain size.

Merge. Similar to split, suppose that a packed object consists of q raw objects, one of which is of size u (see Fig. 3b). The cost for this contained object is therefore

$$C_{merge} = mR_d + \frac{1}{q}R_u + u\Delta t_{max}S + umT_d + uT_u,$$

comparing with (2), $C - C_{merge}$ can be computed as in Formula (5):

$$\begin{cases} (1 - \frac{1}{q})R_u + uS(\Delta t - \Delta t_{max}) & , q > 1, \\ 0 & , q = 1. \end{cases} \qquad (5)$$

Because of the distinct life cycle for each component in a packed object, the object stored in the cloud tends to comprise garbage data. To address this issue, a cleaning process should be considered. Ideally, if all the constituent objects have the same life span (i.e. Δt is equal to the maximum Δt_{max}), the system can discard the whole orphaned object in one transaction. Otherwise, special concerns are required when considering the cost of storing and reclaiming garbage data separately. With the growth of the number of objects (namely increasing q), the cost reduction due to merging multiple PUT requests can approach to R_u. For most cloud vendors, R_u is a dozen times larger than R_d. That means merging has more potential than caching/pre-fetching from the aspect of saving cost. However, considering the unpredictable nature of object life cycle and the complexity of garbage collection, we conclude that effectively exploiting the merging operation is pivotal for Coral, which is demonstrated by our dedication to the design of caching policy and data layout as discussed in the following sections.

Compression. Since the performance bottleneck in Coral is shifted from the hard drive to WAN access, we have more opportunities to employ complex computations such as compression to optimize the storage cost. For example, reducing the unit storage size in the cloud by compression can save the cost by $(\Delta tS + mT_d + T_u)$.

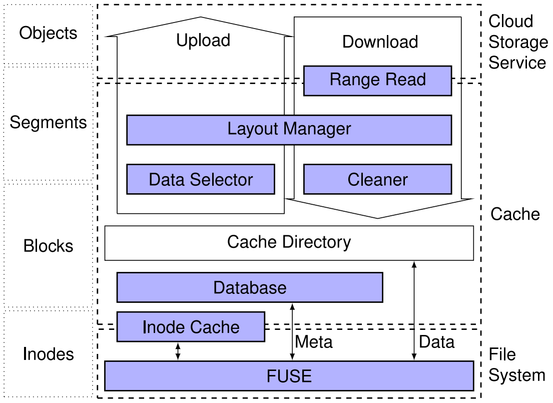


Fig. 4. System architecture of Coral.

3.2 Architectural Overview

Fig. 4 depicts the main components (shaded areas) in Coral. The FUSE [6] layer acts as the interface to intercept read and write requests, monitors the access patterns of applications, and invokes other relevant modules. Specifically, data blocks belonging to different files are stored in the *Cache Directory* that interacts with the cloud storage backend, while the metadata such as the directory structure and file information is maintained in the *Database*, which is periodically synchronized to the cloud. This locally maintained database allows for fast responses to operations that do not need to access file content by avoiding network round trips. Furthermore, we cache the most frequently used metadata of a file (*inode*) to narrow the speed gap between the local file system and storage backend. When the cache eviction is triggered, the *Data Selector* searches for the most unwanted blocks with inherent correlation, reflected by selected metrics and organizes them into *segments*. After being compressed/encrypted, segments are assembled into an *object* by the *Layout Manager*, and eventually uploaded to the cloud. In the event of cache miss or garbage collection, Coral fetches data from the storage backend using HTTP range request, which enables the optimization of downloading unrelated segments independently. Next, unpacked/decrypted blocks are ingested into the cache. The *Cleaner* performs the operation of garbage collection incurred by combining blocks with different life cycles into the same segment. In the following, we describe each component of Coral and cost optimization strategies in detail.

3.3 Metadata Database

Coral manages metadata using the relational *database*. As shown in Fig. 4, four main entities compose the backbone of metadata, including *inodes*, *blocks*, *segments*, and *objects*. They are mapped to database tables directly. For instance, *inodes* are stored in the table with schema *inodes(id, uid, gid, mode, mtime, atime, ctime, size, rdev, locked, refcount)*. In terms of relations, the many-to-many relation requires additional tables to describe, like *inode_blocks(inodeid, blockno, blockid)*, while for the many-to-one relation we can build a linkage by *id* reference (e.g. the table *segments(id, hash, size, offset, activecount, objectid)* with *objectid*). Lastly, some secondary tables are defined for extra file attributes (e.g. symlink).

For modern database systems, this simple schema could be handled effectively, absorbing substantial file system operations locally. However, with an increasing working set in Coral, a natural question is that whether the growth of metadata size is acceptable in our system. Next, we discuss the impact of the schema design on metadata capacity for the entire storage system.

TABLE 2
A Case for Metadata Capacity Estimation (M = 32 KB)

Table	B (B)	I	C	V'	V''	V'''
inodes	124	1.2	0.3	1	2	0.5
blocks	92	1.2	0.3	1	1	1
segments	75	1.1	0.4	0.1	0.1	0.1
objects	46	1	0.4	0.01	0.01	0.01
			S_m (GB)	1	1	1
			S_d (GB)	465.16	293.92	656.36

3.3.1 Capacity Estimation

Since the sizes of the majority of fields in the schema stay fixed, the metadata size S_m can be estimated with the following intuitive approach. For each table j, we calculate S_m, the sum of the multiplication of B_j (the size of all columns in bytes), r_j (the number of estimated rows), I_j (the database index factor), and C_j (the compression factor), as expressed in Equation (6). In practice, the storage engine of the database tends to provide specific features or maintain internal structures, changing the concrete database size. We can introduce more correction coefficients for the equation to improve the estimation

$$S_m = \sum_{j \in tables} B_j \cdot r_j \cdot I_j \cdot C_j. \tag{6}$$

Furthermore, the number of rows in tables have inherent correlations. The number of rows in the table *objects*, for example, is several orders of magnitude smaller than that of the table *inodes*. Also, distinct row numbers among tables reflect different types of workloads. We use the number of rows γ of table *blocks* and a coefficient vector V to represent $r_j = \gamma \cdot V_j$ of all tables, where $V_{blocks} \equiv 1$. In this way, by defining parameters V, we can obtain S_m with respect to γ in Equation (7). Accordingly, the capacity of the working set is $S_d = \gamma \cdot M$, where M denotes the average size of blocks. We can finally deduce the relation between S_m and S_d as shown in Equation (8)

$$S_m = \gamma \sum_{j \in tables} B_j \cdot V_j \cdot I_j \cdot C_j \tag{7}$$

$$= \frac{S_d}{M} \sum_{j \in tables} B_j \cdot V_j \cdot I_j \cdot C_j. \tag{8}$$

3.3.2 Case Study

Table 2 shows a case of estimation. To simplify, we just present four main entities and increase the B slightly for compensating unlisted ones. The selection of I and C is based on the page level index and zlib based compression [17] in the database. Under such parameters, we estimate the working set capacity S_d (shaded cells) by setting $S_m = 1\,\text{GB}$. By varying V to simulate different workloads, we can observe that the metadata size is reasonable in Coral.

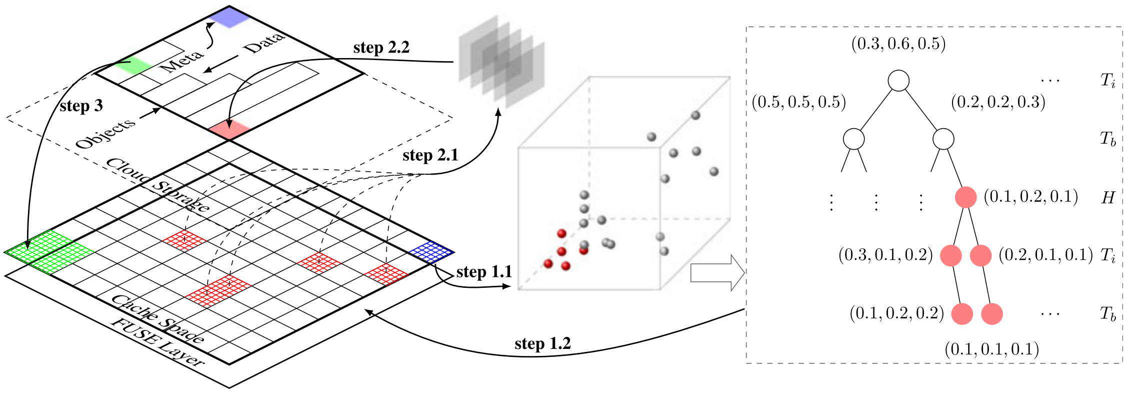


Fig. 5. The main steps performed by the caching subsystem.

For high aggregate storage requirement (e.g. enterprise scenarios), the *Database* component can be independently deployed to serve Terabyte level metadata management. More detailed evaluation is presented in Section 4.2.3.

3.4 Caching Policy

The cache management plays a vital role in the whole system. In this section, we clarify the rationales behind the cache system design from two aspects. First, we discuss the algorithmic designs that reflect our concerns of identifying correlated blocks accurately and quickly to facilitate the decision in evicting cached content. Second, we present a self-adaptive approach to addressing the issue of cache thrashing caused by the radical changes of access patterns.

3.4.1 Correlated Blocks and Data Structures

In Section 2.2, we have demonstrated the potential of exploiting the inter-block relationship to optimize remote data access from a macroscopic view. In this section, we illustrate how the three concrete metrics (inode, block, and hotness) can be used to represent the correlation in a 3D space intuitively, and its superiority in improving cache efficiency. The new strategy has three key advantages. First, not only the hotness of data but the relationship implicitly exhibited by block access patterns is utilized to make flexible decisions on swapping in/out data blocks, as compared to the LRU policy. Second, the original block sequence generated by dividing files is rearranged according to the hotness information, which provides hints for the data layout engine (see Section 3.5). Third, we can balance the trade off between local computation and remote access based on the HTTP range parameter.

However, two major challenges in effectively selecting and grouping blocks need special considerations. On one hand, the established correlation may become invalidated due to the dynamic nature of varying workload, and if not appropriately handled, a phenomenon we call cache thrashing may occur as detailed in Section 3.4.3. On the other hand, although the latency of accessing the cloud can partially mask the overhead incurred by a computation-intensive caching algorithm (as compared to LRU), prompt identification of correlated blocks is critical to the overall performance due to the synchronous nature of file accesses. To this end, We use a kd-tree data structure to maintain the 3D metrics space.

The kd-tree is useful for space partitioning and well-suited for our needs. In particular, each non-leaf node that stores 3D data can be thought of as implicitly generating a splitting hyperplane. By dividing the space into two parts, known as half-spaces, the performance approaches binary search in subtrees. Given a query key, we can obtain corresponding neighbors in certain distance based on fast tree traversal, fulfilling the task of finding correlated blocks. The right part of Fig. 5 depicts a 3D space and the relevant kd-tree.

3.4.2 Algorithmic Design

The caching policy is critical to system performance, and its algorithmic design aims at exploiting the access patterns of data blocks to reduce remote accesses to the cloud and better utilize the cache space. To better understand the cache subsystem, we use an illustrative example as shown in Fig. 5. On the left part, blocks and metadata in the cache space are represented by small squares at the middle layer, and the arrowed lines marked with step indexes indicate the main operations performed in managing the cache subsystem.

Step 1. We first discuss the procedure of evicting cached blocks to make room for new data. As shown in Algorithm 1, the metadata including the last access time of blocks/inodes, the size and access counts for each block in the cache space, is queried from the database (step 1.1). Based on the relative time interval and nonlinear transform, a kd-tree is generated to represent the metrics of data blocks in the 3D space. Generally, the kd-tree is rebalanced to obtain a complete binary tree to maximize search efficiency. We use two parameters *key* and *dist* to get the coldest blocks in the logical 3D space. The candidate *key* selected (red spaures) from the kd-tree reflects not only the hotness but the closeness of data blocks. The parameter *dist* is used to determine the packing degree, a metric to optimize cache thrashing as further discussed in Section 3.4.3. Since the cache subsystem and other components heavily rely on querying the *database*, we design an *inode cache* to accelerate database access, serving the majority of requests.

Step 2. Commonly, when a block is selected as a candidate for eviction, it can be locked to prevent any further accesses. However, the eviction operation consists of several sub-steps such as metadata querying, data merging, and encryption/compression, each demanding different access permissions on data blocks. Imposing exclusive lock on each block during the whole operation is too conservative. For example, before encryption/compression, read access to a block should be granted because there is no conflicts with other operations. In this regard, we employ fine-grained locking on each block to facilitate the multi-threaded design at the backend, which can improve the throughput of the cache.

Algorithm 1. Select (Part 1)

```
$metrics\_set \leftarrow query\_db(blocks\_in\_cache)$
$[min\_inode, delta\_inode] \leftarrow elapse(metrics\_set)$
$[min\_block, delta\_block] \leftarrow elapse(metrics\_set)$
$key \leftarrow (1, 1, 1)$
$tree \leftarrow init\_kdtree()$
for all $m \in metrics\_set$ do
    $ti \leftarrow (m.Ti - min\_inode)/delta\_inode$
    $tb \leftarrow (m.Tb - min\_block)/delta\_block$
    $h \leftarrow 2 \times tan^{-1}(m.N/m.S)/\pi$
    $point \leftarrow (ti, tb, h)$
    $tree.add(point, i)$
    if $key > point$ then
        $key \leftarrow point$
    end if
end for
if $tree.isbalance() = False$ then
    $tree.rebalance()$
end if
                    $\triangleright blocks\_to\_swapout \leftarrow tree.knn(key, \mathbf{dist})$
```

Step 3. Algorithm 2 shows the cache replacement logic in Coral. The data unit downloaded from the cloud is a segment that contains data blocks with correlated relationship, and its structure is detailed in Section 3.5. Segment based data transfer also implies that blocks swapped in can act as pre-fetched data to improve the hit rate of future requests. In addition, we use the feature of HTTP range request (indicated by the parameters *from* and *to*) to precisely control the amount of data remotely fetched, which is marked by green squares for the cloud storage layer in Fig. 5.

Algorithm 2. Fetch

```
$block\_id \leftarrow fuse\_request()$
if $cache.has(block\_id) = False$ then
    $[object\_id, from, seg\_size] \leftarrow query\_db(block\_id)$
    $to \leftarrow from + seg\_size$
    $more \leftarrow MAX\_SIZE - (cache.size() + seg\_size)$
    if $more \geq 0$ then
        ... swap out ...
    end if
    $segment \leftarrow request\_cloud(object\_id, from, to)$
    $blocks \leftarrow split(segment)$
    for all $item \in blocks$ do
        $data \leftarrow decrypt(item)$
        $block \leftarrow decompress(data)$
        $cache.set(block\_id, block)$
    end for
end if
$requested\_block \leftarrow cache.get(block\_id)$
```

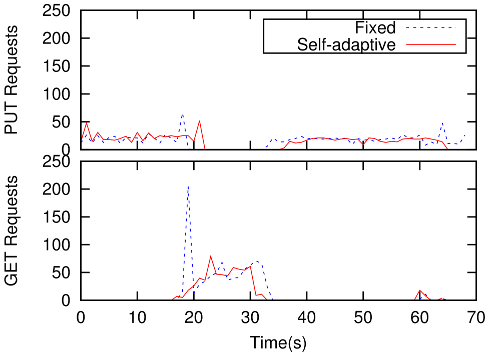


Fig. 6. Effect of the self-adaptive mechanism.

3.4.3 Resolving Cache Thrashing

Certain circumstances may undermine the caching mechanism. When the access mode of the file system changes, the established information for correlated data blocks may have negative impact on performance if it is directly used for making subsequent caching decisions. For example, the Linux kernel compilation benchmark has two stages: file extraction and building. The file extraction stage involves only write operations, and the collected access patterns in this stage provide little help for the following one that exhibits read-write access patterns. At the transitional period of radical workload change, a phenomena called cache thrashing would occur if the caching policy solely relies on the outdated block relationship. To address this issue, we consider three access modes (read-only, write-only, and read-write), and the transition between them is traced at runtime to provide necessary hints for accommodating the dynamically varying workloads.

Our solution depends on the parameter *dist* (see Section 3.4.2) derived from the file access information recorded at the FUSE layer, to adaptively control the packing of data blocks at runtime. More concretely, we use an array of counters that composes a fixed-size window to record the most-recent read/write requests. With this data structure, we can accurately detect the transition of access patterns, and then react with appropriate adjustment to the key parameter *dist*.

As shown in Algorithm 3, we first determine the current access mode by comparing the value calculated from the read/write windows (variables *read* and *write*) with the last mode recorded by *last_mode* (see lines 20-31). Next, the distance parameter (*dist*) that would guide the metadata search in the kd-tree is derived based on the current system status (see lines 33-42). In order to avoid cache thrashing caused by oversized or undersized storage objects during mode transitions, the function *minimum_dist* iteratively search the kd-tree to obtain a desirable value for the distance metric. Given that the mode transition typically lasts for certain period of time instead of a transient event, and that transitions may repeat frequently over a short time window, the self-adaptation is run for a pre-defined duration as indicated by the variable *mode_delay*. At last, it is worth noting that the kd-tree based metadata management makes it equally well suited for both the distance and neighbors based search in the algorithmic design.

We use the Linux kernel compilation benchmark to verify the effectiveness of our proposed strategy. Fig. 6

depicts the number of read/write requests issued to the remote storage backend, which has marked impacts on the cache performance and overall cost. Specifically, read requests begin to dominate after the extraction of source files (write-only). Without self-adaptive optimization, the number of read requests issued to the cloud increases significantly at the start of the transition of access mode (at about the 17th second). In comparison, with the optimization enabled, we can observe the smoothed variation curve over the same time period. This strategy also works for other workloads, although we only present the results for the kernel compilation.

Algorithm 3. Select (Part 2)

```
$[read\_w, write\_w] \leftarrow fuse\_windows()$
$read \leftarrow sum(read\_w); write \leftarrow sum(write\_w)$
$mode \leftarrow NO\_OPS$
if $read \neq 0$ or $write \neq 0$ then
    $mode \leftarrow READ\_WRITE$
    if $read = 0$ and $read = last\_read$ then
        $mode \leftarrow WRITE\_ONLY$
    end if
    if $write = 0$ and $write = last\_write$ then
        $mode \leftarrow READ\_ONLY$
    end if
end if
$last\_read \leftarrow read; last\_write \leftarrow write$
if $mode\_delay > 0$ then
    $dist \leftarrow minimum\_dist()$
    $mode\_delay \leftarrow mode\_delay - 1$
else
    if $last\_mode = mode$ then
        $dist \leftarrow normal\_dist()$
    else
        $mode\_delay \leftarrow DELAY\_COUNT$
    end if
end if
$last\_mode \leftarrow mode$
$blocks\_to\_swapout \leftarrow tree.knn(key, dist)$
```

3.5 Data Layout

Given that the design of the data layout is driven by the requirements from several factors such as the storage and transfer cost, the interoperability with the cache subsystem, in this section, we first present how data blocks is organized and linked together. Then, we address the issue of how to reclaim the aged data in the cloud, which is an inevitable side-effect due to the design trade-offs aforementioned.

3.5.1 Segment Structure

At the core of the caching policy is the capability of identifying correlated data blocks to be evicted from the cache. To facilitate data management, the inherent relationship among data blocks indicated by the caching subsystem should be reflected by how they are organized and persisted in the cloud. The mainstream cloud services do not specify the concrete format for storage objects, and provide support for almost unlimited object size (from 1 KB to 5 TB in S3). And unlike the conventional storage stack, existing objects in the cloud cannot be updated directly without uploading new substitutions for old copies. Coupled with the design choices of the caching policy, this limitation motivates a design of data layout that enables optimized read/write performance and reduced cost incurred by data requests and garbage collection.

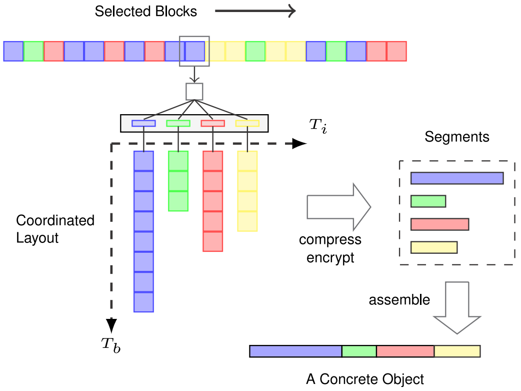


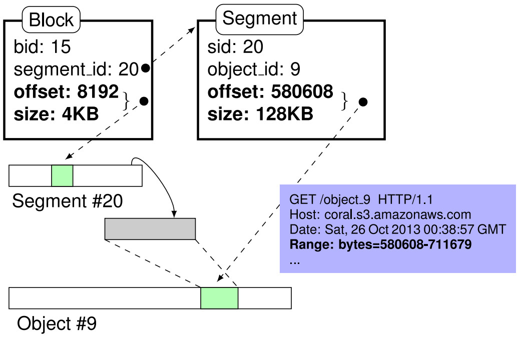
Fig. 7. Data layout of a concrete storage object.

Fig. 7 illustrates the segment structure. First, the selected blocks are arranged successively into two-dimensional layout that exhibits the correlation of files (inode) and blocks. This layout is merely a conceptual illustration, and we do not physically maintain such a structure in the memory because it will result in unnecessary memory copies when these blocks are composed into segments. Second, blocks belong to the same inode compose a single segment, and are operated as an independent unit for compression and encryption. Third, we assemble all the segments according to the relationship implied by the dimension T_i to produce a concrete storage object.

In this way, PUT requests (relatively expensive) are managed at the granularity of objects, while cheaper and frequent read (request and transfer) operations retrieve data by segments. The distinguishable access granularity employed in remote read/write request is helpful to save cost according to the billing model (Section 2.1). We next detail how to fetch data from the cloud.

3.5.2 Offset Remapping

Since the location information of data blocks would be lost due to the operation of compression and encryption performed on segments, we can not obtain the correct offsets in storage objects for the requested blocks directly. We therefore remap the block offset to the segment offset that is used as the range parameter of HTTP request. Fig. 8 depicts two pairs of (*offset*, *size*) in a block and its container segment respectively. These metadata are stored in the database when the segment structure is constructed. For requesting the block with *bid* 15, we first need to know which segment the requested block belongs to. With the *segment_id* (20) acquired from querying the database, another query is issued to obtain the *offset* and *size* of the segment indicated by the *segment_id*, which are then used to calculate the value of the byte range for the *Range* header. Finally, after the specified segment is downloaded from the cloud, the target blocks can be extracted.



```
GET /object_9  HTTP/1.1
Host: coral.s3.amazonaws.com
Date: Sat, 26 Oct 2013 00:38:57 GMT
Range: bytes=580608-711679
...
```

Fig. 8. Offset remapping for block request.

3.5.3 Garbage Collection

The segment structure reflects the inherent relationship among blocks, which is beneficial to the optimization of data accesses by lowering the possibility of consolidating completely independent blocks in one storage object. However, complex and dynamic runtime behaviors of applications inevitably induce distinct life cycle on data blocks, and aged blocks should be discarded in a cost-effective way. For example, blocks being updated (e.g. editing a document) can not instantly replace the counterparts in the cloud, instead we have to retain old copies because of the constrains of cloud service model. Aged data will incur not only extra storage expense but also data transfer overhead due to the defragmented segment structure. In the following, we address this issue by answering when and how to perform the operation of garbage cleaning.

Scheduling garbage collection should take into account the cost involved in two aspects. On one hand, the storage cost denoted by $C_{store}(t)$ will accumulate from each object with aged blocks as shown in Equation (9). Inside an object, C'_i is determined by the size of all aged blocks s_j and the difference between the time t^0_j when the block are marked as deletable and the time t when the cleaner is started, as shown in Equation (10). On the other hand, the cost incurred by the cleaning operation itself (represented by C_{clean}) should also be counted. To this end, we prove the optimal time instance of issuing the cleaning operation as shown in Theorem 1

$$C_{store} = \sum_{i \in objects} C'_i \tag{9}$$

$$= \sum_{i \in objects} \sum_{j \in aged} s_j \cdot (t - t^0_j). \tag{10}$$

Theorem 1. *The cleaning time τ is optimal when the Eq. (11) holds, which indicates that the cost caused by garbage collection only depends on $C_{store}(\tau)$*

$$C_{store}(\tau) = C_{clean}. \tag{11}$$

Proof. The desirable cleaning scheme should minimize the cost $C_{store}(t) + C_{clean}$. With the arithmetic and geometric means inequality, we have

$$C_{store}(t) + C_{clean} \geq 2\sqrt{C_{store}(t) C_{clean}}.$$

The equality holds true if and only if there exists τ such that $C_{store}(\tau) = C_{clean}$, from which we can induce the best time to start the cleaning operation. In addition, the total cost $2\sqrt{2C_{store}(\tau)}$ is independent of any specific cleaning algorithms. □

Moreover, the equality also implies the opportunity of performing direct and convenient comparison between different cleaning approaches, since the cost of disposing aged data is always computable. We next analyze two particular cases.

In Coral, we implement the cleaner as an integrated component of our system that interacts with the block management and caching subsystem to fulfill the task of space reclamation. The C_{clean} can be calculated using Equation (12), where T_d, R_u, and R_d denote the unit price defined in Section 3.1.1, and S_i represents the object size. Assume we have an object to edit, after downloading the corresponding segments without aged blocks (range to the greatest extent) by n requests, the cleaner reconstructs and then uploads the new objects to the cloud at certain time. The upstream transfer and delete operation in the cloud are free. According to Theorem 1, the cleaning operation will be launched at the time when $C_{store} = C_{clean}$. We will evaluate garbage cleaning in Section 4.3.3

$$\begin{aligned} C_{clean} &= \sum_{i \in objects} C''_i \\ &= \sum_{i \in objects} (T_d \cdot (S_i - \sum_{j \in aged} s_j) + nR_d + R_u). \end{aligned} \tag{12}$$

Secondly, It is possible to locate the cleaner at the same place as the storage backend, and the main benefit of this option is the intra-datacenter data access that can expedite data processing and avoid the extra cost of upstream and downstream data transfer. Assume that the cleaner is delegated to a compute node in the cloud such as the EC2 [1]. Then we have

$$C_{clean} = W \cdot t',$$

where W denotes the unit price of the compute node, and t' is the execution time of the cleaning operation depending on the volume of aged data. In this case, we can also start the cleaning routine under the guidance of Theorem 1. Therefore, distinct cleaners become comparable via C_{store}. We plan to incorporate the advantages of the standalone cleaner design in future work.

3.6 Implementation Issues

Our current implementation of Coral contains approximately 7,500 lines of Python code, and a few hundred lines of C++ in performance-critical components, including the kd-tree selector and delta-dump/restore functions for the database. Coral is built on top of FUSE that supports standard Unix file system features, and uses the *sqlite* database to maintain metadata. With an extensible architecture, Coral provides support for multiple storage backends including Amazon S3 (or compatible protocols), Google Cloud Storage, and local disk, and several representative caching schemes are implemented for comparison. All data is compressed using the LZMA algorithm [10] and encrypted using AES [7] with a 256-bit key.

Data consistency. Current cloud storage systems typically sacrifice strong consistency in favor of performance and availability, as a result, only eventual consistency is guaranteed. Under this model, operations such as read-after-write may occasionally fail, and the responsibility to tolerate such failures is imposed on the developers consequently. Other client situations such as machine crashing also might lead to data inconsistency. To address this issue, we assign each block or object an ID that is automatically increased when it is updated. Further, the metadata (database files) is associated with a version number, and when the two version numbers do not match, a series of measures would be conducted to ensure data consistency. Coral uses the latest-version-wins mechanism to resolve conflicts based on version numbers, which means the newest version is valid by default. Meanwhile, several old versions are also maintained temporarily, providing the opportunity of rolling back to a specific version selected by the user.

Snapshot. In Coral, the creation of snapshot is straightforward and incurs little overhead, since the metadata, including the information about files and the directory structure, are recorded in the database. We can conveniently preserve the state of the entire file system at any time using a minimum amount of storage space.

4 Evaluation

In this section, we evaluate the Coral prototype implementation with macro- and micro-benchmarks. Our experiments were conducted in a cross-continent environment. In addition to perform the comparison with the state-of-the-art using real world applications and synthetic benchmarks, we also investigate the performance advantage of our caching policy and the effect of garbage collection on run time and storage cost.

4.1 Setup

We used the Linode 2048 Plan (at the Japan datacenter) [9] running Ubuntu 12.04 LTS as the client to launch the benchmarks. The instance configuration was a 64-bit platform with 2 GB RAM, 2.27 GHz Intel Xeon L5520 (quad-core) processors, and 80 GB storage. Our experimental measurements were performed with Amazon S3 storage backend in two regions, N. Virginia (US-East) and N. California (US-West). For testing the network connectivity between Linode (Japan) and Amazon (US), we used an EC2 node located at the same datacenter as S3 to run *iperf*. The average bandwidth between the client and S3 in US-West and US-East reached 21 and 34 Mbps respectively.

Our benchmarks include compiling the Linux kernel and generating synthetic workloads with Filebench. The workloads produced by the first one contain extracting the source code of Linux kernel 3.4.4, which consists of roughly 40,000 files totaling 450 MB (write-only workload), calculating the checksum of all these source files (read-only workload), and building the kernel with the default configuration and *j4* flag for parallel compilation (read-write workload). Filebench has various prepackaged workload personalities, among which fileserver, webserver, varmail and netsfs are used.

4.2 Macro-Benchmarks

We first examine our goals on optimizing monetary cost and latency, by comparing with Bluesky that is characterized by a log-structured data layout with an LRU caching policy. In Bluesky, write requests are collected into a fixed size segment (4 MB).

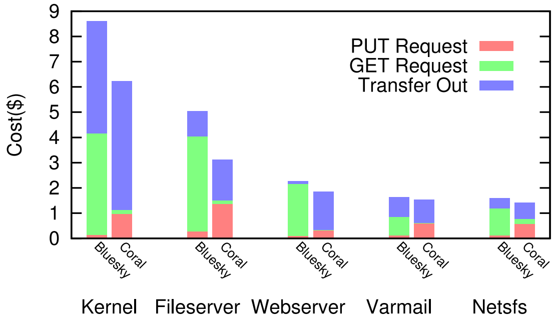


Fig. 9. Evaluation results of monetary cost.

4.2.1 Cost

Cloud storage providers typically charge customers over a monthly billing cycle. In our scenario, the storage cost is relatively small because of the limited time window of running the experiments. However, the cost caused by data transfers and requests is much more prominent in file system workloads. Thus, we only analyze the fees actually charged for the later two metrics, excluding the storage cost. To guarantee accuracy, our analysis is based on S3's internal logging tool that generates the access log and stores the log files as normal storage objects asynchronously in the cloud. The prices in October 2013 from Amazon S3 US-East region are: $0.095/GB per month, $0.12/GB transfer out, and 0.005 per 10,000 GET or 1,000 PUT operations (base prices).

Fig. 9 compares the monetary cost between Bluesky and Coral. We consider three billing items: PUT/GET request and outbound data transfer from the cloud. The average improvement with Coral is about 28 percent over all applications, among which the fileserver and kernel complication achieve better cost savings, whereas in scenarios where read request dominates such as webserver, varmail, and netsfs, Coral exhibits similar results as Bluesky. Moreover, we can observe the increased cost of two billing items. First, the data transfer cost in Coral is more prominent due to the prefetching of interrelated data blocks when downloading data from the cloud (see the webserver case). Second, compared with the fixed-size storage objects as in Bluesky, our system generates variable-size objects due to the need of tracking the mode switches, which leads to the increase of PUT requests. Even with the increase of the cost of data transfer and PUT requests, the total cost of Coral is lower than Bluesky across all benchmarks because of the significantly reduced GET requests. We are investigating new ways to further reduce the cost incurred by data transfer and PUT requests.

4.2.2 Impact of Latency

We use the same benchmarks as the above to underscore the impact latency can have on file system performance. Particularly, measuring the kernel compilation time is decomposed into three steps (unpack, checksum, and build) to reveal the results for distinct access modes. We ran this experiment against both the local and cloud storage.

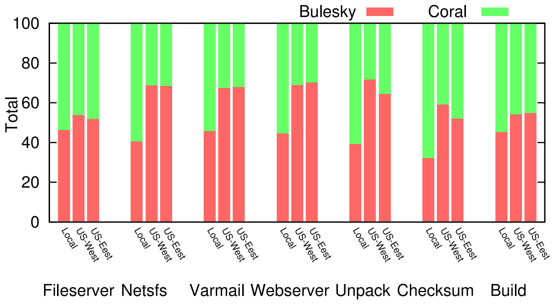


Fig. 10. Excution time comparison under various workloads.

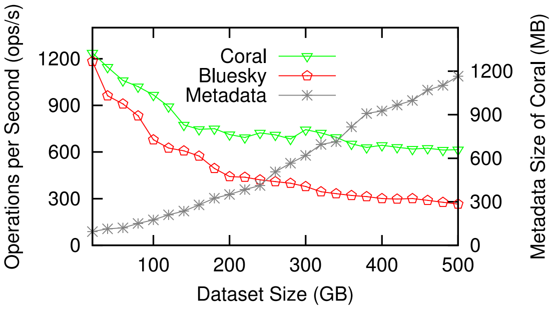


Fig. 11. Operations per second of distinct dataset sizes.

Fig. 10 shows the comparison of normalized running times, which is more intuitive than the illustration of plotting the absolute timing results that exhibit relatively large variation for different workloads. So, a 50:50 split means equal times for Coral and Bluesky. For local storage, Bluesky consumes less time (about 26 percent averagely) across all benchmarks. In comparison, Coral performs better than Bluesky with the cloud storage as backend, and the average improvements are about 83 and 69 percent for the two backends (US-West and US-East) respectively. The log-structured design in Bluesky originally aims to improve the write performance of hard disk by organizing random writes sequentially with a log. However, this advantage may have limited impact on performance improvement for cloud-backed file systems, because the underlying performance-critical factors have changed. As evidenced by the comparison between the two geographical locations, with better network connectivity to our client machine, the US-West region has shown better results. Moreover, for Coral, benchmarks dominated by a single operation (read or write) such as netsfs, varmail, webserver, checksum, and unpack exhibit better performance than others that involve more access mode switches, we attribute this to the overhead incurred by the self-adaptive intervention.

4.2.3 *Large Workload*

Existing studies [37], [39] found that the average amount of accessed data per day is only 10 percent of the total dataset in a typical networked file system, indicating that Coral can serve hundreds TB data with a tens TB local cache. In this section, we evaluate the performance of Coral under large workload. To this end, we use the fileserver personality in Filebench to simulate operations on a relatively large dataset including *create*, *delete*, *append*, and *read* operations on files and directories. This is similar to the workloads generated by SPECsfs as used in Bluesky. We set the cache size to 10 GB with the storage backend at S3 US-West, and use operations per second (ops/s) as the metric to compare the performance between Coral and Bluesky. In addition, the capacity of the metadata database is also evaluated to validate our theoretical analysis. As shown in Fig. 11, with the increasing of dataset size, we can observe the decreasing performance for both Coral and Bluesky because the fixed-size cache necessitates more remote data accesses. But the performance of Coral stays better than Bluesky across all sizes of dataset. Notably, the volume of metadata produced by this large workload is at roughly the same level as the theoretical estimation in Section 3.3. According to the results, we believe our system can scale to large workloads. In real enterprise scenarios, a dedicated high-performance machine can be deployed to manage even higher volume metadata.

4.3 Micro-Benchmarks

In this section, we first demonstrate the superiority of our caching strategy compared to the traditional LRU policy, and then present the evaluation on garbage collection.

4.3.1 *Cache Performance*

This experiment was run with cache capacity varying from 2 to 20 percent of the dataset size, to evaluate how the cache hit ratio scales under the two caching policies.

Fig. 12 illustrates the cache hit ratio of LRU and KD over the four Filebench applications and Linux kernel compilation. For all benchmarks, the hit ratio increases with the increase of cache capacity. In particular, the cache hit ratio of netsfs and varmail is higher (over 60 percent) than that of the webserver, fileserver and Linux, because of the relatively more randomness in I/O requests. By comparing the variations of cache hit ratio of KD and LRU policy, we can observe that the KD policy shows more stable fluctuation across all cache sizes. For example, in the case of netsfs, the hit ratio of LRU ranges from 64 to 88 percent, while with KD it varies from 84 to 92 percent, showing smaller

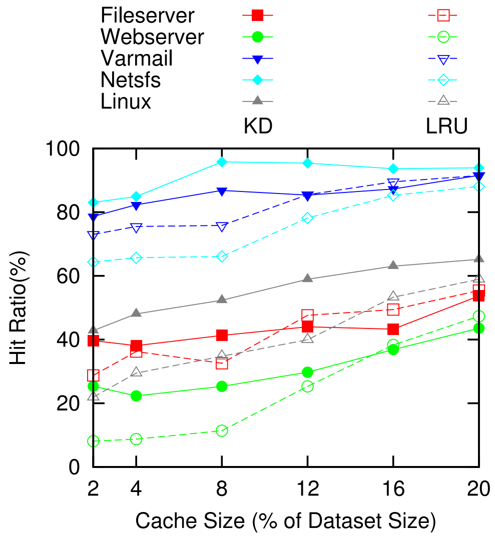


Fig. 12. Cache hit ratios of distinct workloads and datasets.

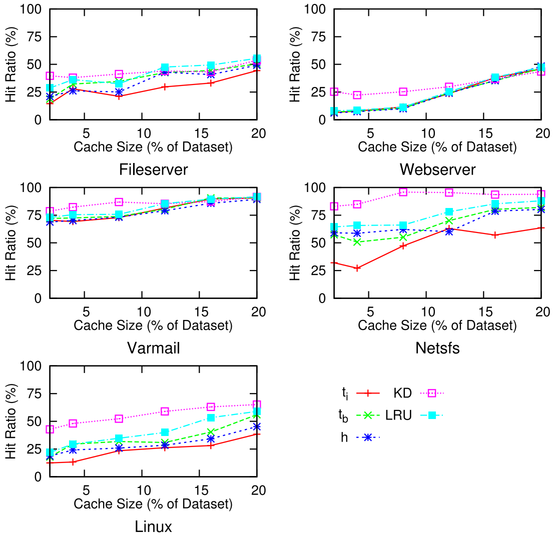


Fig. 13. Cache hit ratios for individual metrics.

magnitude of fluctuation. Other benchmarks exhibit similar trends. This stability has important implications for cloud-backed file systems, because of the potentially unlimited cloud storage that far exceeds the cache size on the client side. Taking advantage of the data block relationship, as expected, the KD policy achieves higher hit ratio than LRU for most cache sizes. However, diminishing returns of the KD policy are observed for certain workloads with the increasing of cache size, because of the decreased exploitable interrelated data blocks.

4.3.2 Impact of Individual Metrics on Cache Performance

In this section, we evaluate the impact of individual metrics (t_b, t_i, and h) on cache hit ratio, and use the same workloads and configurations as the above section. In addition, we also duplicate the results in Fig. 12 for intuitive comparison. As shown in Fig. 13, the KD policy performs better than any single metric. We analyze the performance of individual metrics as follows. First, the metric t_b only contains the information of the time dimension (no frequency) in LRU, thus exhibiting worse cache hit ratios than LRU. Second, for the metric t_i that is similar to file-based (not block) LRU policy without considering the frequency, we can observe similar performance results between t_i and t_b for workloads (e.g. webserver and varmail) whose average file size approximates to the block size. While for other workloads with large amount of blocks per file, the cache hit ratios of t_i are lower than that of t_b, because of behavioral diversity among files and blocks. Third, when using the hotness metric h, the performance varies between LRU and t_i. In summary, the KD policy achieves better cache performance by incorporating the advantages of multi-dimension information. In addition to cache performance, the metrics such as t_i also have implications on choosing parameters for the layout of storage objects.

4.3.3 Cleaning

It is difficult to directly evaluate and compare different garbage cleaning designs. First, distinct cleaning approaches typically involve complex interactions with other modules in the system. For example, Bluesky use an Amazon EC2 node to collect garbage data. Second, calculating the cleaning cost requires a non-trivial amount of garbage data generated by *delete/edit* operations, but our current workloads contain too few such operations to produce the desired experimental dataset. However, as demonstrated by the theoretical analysis in Section 3.5.3, we can measure the cost of garbage collection indirectly based on the storage cost of garbage data. Consequently, we use a synthetic workload generated by Filebench to conduct the following analysis. The file system was populated with 10 file sets (a group of related files), each of which contains eight files totaling 64 MB of data. The benchmark rewrites 50 percent of data (320 MB) with 32 KB I/O block in a multi-threaded environment.

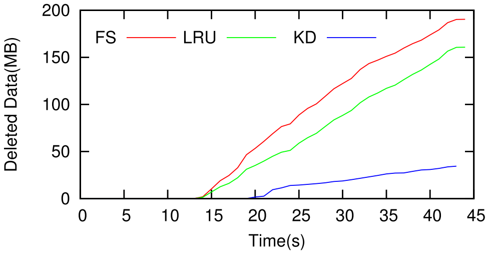


Fig. 14. Comparison of delete requests issued to the cloud.

Fig. 14 depicts the amount of deleted data during the experiment. We use the statistics collected from the file system layer as the baseline (marked as FS), because the file system will perform the same set of operations given the same random seed no matter what caching policy is employed. After the initial stage that populates the file system, the KD and LRU policy exhibit dramatically different behaviors. Since the KD strategy exploits the inherent relationship among data blocks, a significant amount of delete operations are absorbed by the cache, leading to sustained improvement of greater than 60 percent against the LRU policy. More data deleted in the cache means less garbage data stored in the cloud. According to our theoretical analysis, this also implies the reduced cost of performing the cleaning operation in the cloud (less data repackage plus less DELETE requests). Our design principle focuses on eliminating the garbage data stored in the cloud as much as possible, instead of relying on a complicated garbage collection mechanism.

Fig. 15 depicts the varying quantity of garbage data produced in our experiment. Without the mechanism of reclaiming aged data, the total amount of garbage data increases linearly and reaches about 30 MB at the end. In stark comparison, when the *cleaner* is enabled, the garbage

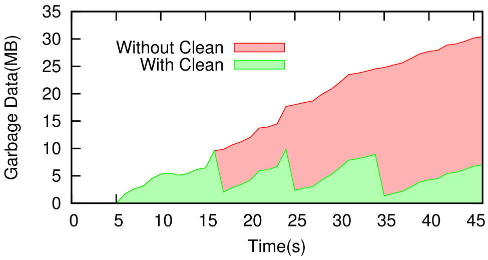


Fig. 15. Effect of garbage collection.

data retained in the cloud is significantly reduced and never exceeds 10 MB. We can observe that the four green triangles have approximately the same size and shape, indicating that the cleaning time is optimally scheduled to minimize the overall cost as discussed in Section 3.5.3. In addition, not all orphaned data are reclaimed each time, which can be inferred from the figure where the green triangular regions are not entirely enclosed. This is because for some cases, live data blocks constitute a dominant part of the storage objects, and it is not economical to remotely manipulate these objects.

5 Related Work

Cloud storage driven by the availability of commodity services from Amazon S3 and other providers has attracted wide interests in both academia and industry. As a new option of storage backend, it greatly eases storage management but brings new challenges as well. The work in this paper reflects our endeavors to address the interrelated issues such as caching strategy, object organization, and monetary cost optimization in designing a cloud based file system.

Cloud based storage. For the service provider, Depot [32] considers safety and liveness guarantees based on distributed sets of untrusted servers. For safety and security reasons, SafeStore [27], DepSky [20] and SCFS [21] operate on diverse storage service providers to prevent cloud vendor lock-in. Recently, Bluesky [38] presents a cloud based storage proxy for enterprise environments. With the log-structured [36] data layout, the proxy can absorb massive remote write requests to achieve high-throughput. However, the design proposed by Blueksy does not consider the billing model that is a major distinct feature compared to classical storage model. Exemplar open source projects like S3FS [14], S3QL [15], and SDFS [12] provide the file system interface for the cloud backend like Amazon S3. Similar to Bluesky, commercial systems such as Cirtas [3], Nasuni [11], and TwinStrata [16] act as cloud storage gateways for enterprises rather than personal users.

Metadata. In [18], the authors reported metadata characteristics over 60 K PC file systems in a large corporation during five-year period. The changes in file size, file type, file age, storage capacity and consumption etc. motivated the database design for metadata management in our work. Also, the study [34] analyzed the whole-file versus block-level elimination of redundancy, which reflects the necessity of block-level deduplication. A recent work [25] showed the behaviors of a popular personal cloud storage service Dropbox [4]. One of the main findings is that the performance of Dropbox is driven by the distance between clients and storage data centers. That corresponds to our design principle on bottleneck shifting to the network.

Storage management. To close the widening semantic gap between computer system and storage system, [33] proposed an classification architecture for disk I/O. The same class of objects (e.g., large files, metedata, and small files) are combined according to the caching policy, which boosts end-to-end performance. Similarly, BORG [22] performs automatic block reorganization based on observed I/O workloads, including non-uniform access frequency distribution, temporal locality, and partial determinism in non-sequential accesses. With reducing the monetary cost as one of the major concerns, we exploit the semantic information among data blocks in all aspects when designing Coral.

Hystor [23] describes a high-performance hybrid storage system with solid state drive (SSD) based on active monitoring of IO access patterns at runtime. HRO [31] treats SSD as a bypassable cache to hard disks. By estimating the performance benefits based on history access patterns, the system can maximize the utilization of SSD.

Cache policy. Beyond typical LRU mechanism, SEER [28] improves cache performance using the sequence of file accesses for measuring the relationship among files. The follow-up study [29] discussed various semantic distances of files and designed an agglomerative algorithm in disconnected hoarding scenario. Also, [30] proposed a two-fold mining method of block correlation mainly based on frequent sequences. In contrast, Coral describes additional eviction basis (besides hotness of data) with temporal-based metric of the file and block, which is simple and easy to measure in file system at runtime with cloud backend ranther than offline mining.

Cost optimization. Few research has specifically studied the issue of monetary cost optimization for the cloud. Chen [24] evaluate cloud storage costs from economic perspective, which is at more abstract level and may not be suitable for complex usage scenarios. In [35], the authors present a system called FCFS with the main focus on the monetary cost reduction for cloud based file systems. However, this work focuses on the optimization for scenarios integrating multiple cloud storage services with distinct cost and performance characteristics. In fact, Coral is complementary to FCFS since optimizing the cost for a single cloud storage service can also benefit systems like FCFS.

6 Conclusions and Future Work

This paper presents the design, implementation and evaluation of Coral, a cloud based file system specifically designed for cloud environments in which improving performance and monetary cost are both principally important for end users. With the efficient data structures and algorithmic designs, Coral achieves our goals of high performance and cost-effective. In the future, we plan to investigate new ways to further reduce the storage cost. For example, using byte-addressable compression algorithms, we can precisely control how much data the client needs to download instead of fetching a complete segment each time.

Acknowledgments

This research was supported in part by the National Natural Science Foundation of China under grants 61272190 and 61173166, the Program for New Century Excellent Talents in University, and the Fundamental Research Funds for the Central Universities of China. Jianhua Sun is the corresponding author.

References

[1] Amazon EC2 Cloud [Online]. Available: http://aws.amazon.com/ec2/, 2015.

[2] Amazon Simple Storage Service (S3) [Online]. Available: http://aws.amazon.com/s3/, 2015.

[3] Cirtas Bluejet Cloud Storage Controllers [Online]. Available: http://www.cirtas.com/, 2013.

[4] Dropbox [Online]. Available: http://www.dropbox.com/, 2015.

[5] Filebench [Online]. Available: http://filebench.sourceforge.net/, 2015.

[6] Filesystem in Userspace [Online]. Available: http://fuse.sourceforge.net/, 2015.

[7] FIPS PUB 197, Advanced Encryption Standard (AES) [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2014.

[8] Google Cloud Storage [Online]. Available: https://cloud.google.com/products/cloud-storage, 2015.

[9] Linode VPS [Online]. Available: http://www.linode.com/, 2015.

[10] LZMA Compression Algorithm [Online]. Available: http://www.7-zip.org/sdk.html, 2015.

[11] Nasuni: The Gateway to Cloud Storage [Online]. Available: http://www.nasuni.com/, 2015.

[12] OpenDudep [Online]. Available: http://www.opendedup.org/, 2015.

[13] Rackspace Cloud Files [Online]. Available: http://www.rackspace.com/cloud/files/, 2015.

[14] s3fs [Online]. Available: https://code.google.com/p/s3fs/, 2014.

[15] S3QL [Online]. Available: https://code.google.com/p/s3ql/, 2013.

[16] TwinStrata [Online]. Available: http://www.twinstrata.com/, 2013.

[17] ZIPVFS: An SQLite Extension For Compressed Read/Write Databases [Online]. Available: http://www.sqlite.org/zipvfs, 2015.

[18] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," in *Proc. 7th Conf. File Storage Technol.*, 2007, pp. 31–45.

[19] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[20] A. N. Bessani, M. P. Correia, B. Quaresma, F. André and P. Sousa, "DepSky: Dependable and decure storage in a cloud-of-clouds," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 31–46.

[21] A. N. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin, and P. Veríssimo, "SCFS: A shared cloud-backed file system," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 169–180.

[22] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reorganization for self-optimizing storage systems," in *Proc. 7th Conf. File Storage Technol.*, 2009, pp. 183–196.

[23] F. Chen, D. A. Koufaty, and X. Z. 0001, "Hystor: Making the best use of solid state drives in high performance storage systems," in *Proc. Int. Conf. Supercomput.*, 2011, pp. 22–32.

[24] Y. Chen and R. Sion, "To cloud or not to cloud?: musings on costs and viability," in *Proc. ACM Symp. Cloud Comput.*, 2011, pp. 29–7.

[25] I. Drago, M. Mellia, M. M. Munafò, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: Understanding personal cloud storage services," in *Proc. Internet Meas. Conf.*, 2012, pp. 481–494.

[26] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, Apr. 2005, pp. 35–35.

[27] R. Kotla, L. Alvisi, and M. Dahlin, "SafeStore: A durable and practical storage system," in *Proc. USENIX Annu. Tech. Conf.*, 2007, pp. 129–142 .

[28] G. H. Kuenning, "The design of the SEER predictive caching system," in *Proc. Workshop Mobile Comput. Syst. Appl.*, 1994, pp. 37–43.

[29] G. H. Kuenning and G. J. Popek, "Automated hoarding for mobile computers," *ACM SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 264–275, Dec. 1997.

[30] Z. Li, Z. Chen, and Y. Zhou, "Mining block correlations to improve storage performance," *ACM Trans. Storage*, vol. 1, no. 2, pp. 213–245, May 2005.

[31] L. Lin, Y. Zhu, J. Yue, Z. Cai, and B. Segee, "Hot random off-loading: A hybrid storage system with dynamic data migration," in *Proc. Simulation Comput. Telecommun. Syst.*, pp. 318–325.

[32] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Trans. Comput. Syst.*, vol. 29, no. 4, pp. 1–38, Dec. 2011.

[33] M. Mesnier, F. Chen, T. Luo, and J. B. Akers, "Differentiated storage services," in *Proc. 23rd ACM Symp. Oper. Syst. Principles*, 2011, pp. 57–70.

[34] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Trans. Storage*, vol. 7, no. 4, pp. 14–20, 2012.

[35] K. P. N. Puttaswamy, T. Nandagopal, and M. S. Kodialam, "Frugal storage for cloud file systems," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 71–84.

[36] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

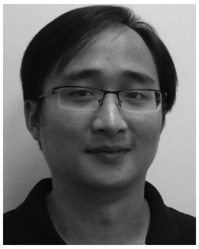
[37] C. Ruemmler and J. Wilkes, "A trace-driven analysis of disk working set sizes," HP Labs, Palo Alto, CA, USA, HP Tech. Rep. HPL-93-23, 1993.

[38] M. Vrable, S. Savage, and G. M. Voelker, "BlueSky: A cloud-backed file system for the enterprise," in *Proc. 7th Conf. File Storage Technol.*, 2012, p. 19.

[39] T. M. Wong and J. Wilkes, "My cache or yours? making storage more exclusive," in *Proc. General Track Annu. Conf. USENIX Annu. Tech. Conf.*, 2002, pp. 161–175.

Cheng Chang is currently working toward the PhD degree in the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests include distributed storage system and virtualization. He is a student member of the IEEE.

Jianhua Sun received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2005. She is an associate professor in the College of Computer Science and Electronic Engineering, Hunan University, China. Her research interests are in security and operating systems. She has published more than 50 papers in journals and conferences, such as the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers*.



Hao Chen received the BS degree in chemical engineering from Sichuan University, China, in 1998, and the PhD degree in computer science from Huazhong University of Science and Technology, China in 2005. He is currently a professor in the College of Computer Science and Electronic Engineering, Hunan University, China. His current research interests include parallel and distributed computing, operating systems, cloud computing and systems security. He published more than 60 papers in journals and conferences, such as the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, IPDPS, IWQoS, HiPC, and CCGrid. He is a member of the IEEE and the ACM.