BlueSky: A Cloud-Backed File System for the Enterprise

Michael Vrable,* Stefan Savage, and Geoffrey M. Voelker

Department of Computer Science and Engineering
University of California, San Diego

Abstract

We present BlueSky, a network file system backed by cloud storage. BlueSky stores data persistently in a cloud storage provider such as Amazon S3 or Windows Azure, allowing users to take advantage of the reliability and large storage capacity of cloud providers and avoid the need for dedicated server hardware. Clients access the storage through a proxy running on-site, which caches data to provide lower-latency responses and additional opportunities for optimization. We describe some of the optimizations which are necessary to achieve good performance and low cost, including a log-structured design and a secure in-cloud log cleaner. BlueSky supports multiple protocols—both NFS and CIFS—and is portable to different providers.

1 Introduction

The promise of third-party "cloud computing" services is a trifecta of reduced cost, dynamic scalability, and high availability. While there remains debate about the precise nature and limit of these properties, it is difficult to deny that cloud services offer real utility—evident in the large numbers of production systems now being cloud-hosted via services such as Amazon's AWS and Microsoft's Azure. However, thus far, services hosted in the cloud have largely fallen into two categories: consumer-facing Web applications (e.g., Netflix customer Web site and streaming control) and large-scale data crunching (e.g., Netflix media encoding pipeline).

Little of this activity, however, has driven widespread outsourcing of enterprise computing and storage applications. The reasons for this are many and varied, but they largely reflect the substantial inertia of existing client-server deployments. Enterprises have large capital and operational investments in client software and depend on the familiar performance, availability and security characteristics of traditional server platforms. In essence, cloud computing is not currently a transparent "drop in" replacement for existing services.

There are also substantive technical challenges to overcome, as the design points for traditional client-server applications (e.g., file systems, databases, etc.) frequently do not mesh well with the services offered by cloud providers. In particular, many such applications are designed to be bandwidth-hungry and latency-sensitive (a reasonable design in a LAN environment), while the remote nature of cloud service naturally increases latency and the cost of bandwidth. Moreover, while cloud services typically export simple interfaces to abstract resources (e.g., "put file" for Amazon's S3), traditional server protocols can encapsulate significantly more functionality. Thus, until such applications are redesigned, much of the latent potential for outsourcing computing and storage services remains untapped. Indeed, at \$115B/year, small and medium business (SMB) expenditures for servers and storage represent an enormous market should these issues be resolved [9]. Even if the eventual evolution is towards hosting all applications in the cloud, it will be many years before such a migration is complete. In the meantime, organizations will need to support a mix of local applications and use of the cloud.

In this paper, we explore an approach for bridging these domains for one particular application: network file service. In particular, we are concerned with the extent to which traditional network file service can be replaced with commodity cloud services. However, our design is purposely constrained by the tremendous investment (both in capital and training) in established file system client software; we take as a given that end-system software will be unchanged. Consequently, we focus on a proxy-based solution, one in which a dedicated proxy server provides the *illusion* of a single traditional file server in an enterprise setting, translating requests into appropriate cloud storage API calls over the Internet.

We explore this approach through a prototype system, called BlueSky, that supports both NFS and CIFS network file system protocols and includes drivers for both the Amazon EC2/S3 environment and Microsoft's Azure. The engineering of such a system faces a number of design challenges, the most obvious of which revolve around performance (i.e., caching, hiding latency, and

*Current affiliation: Google. The work in this paper was performed while a student at UC San Diego.

maximizing the use of Internet bandwidth), but less intuitively also interact strongly with cost. In particular, the interaction between the storage interfaces and fee schedule provided by current cloud service providers conspire to favor large segment-based layout designs (as well as cloud-based file system cleaners). We demonstrate that ignoring these issues can dramatically inflate costs (as much as 30× in our benchmarks) without significantly improving performance. Finally, across a series of benchmarks we demonstrate that, when using such a design, commodity cloud-based storage services can provide performance competitive with local file servers for the capacity and working sets demanded by enterprise workloads, while still accruing the scalability and cost benefits offered by third-party cloud services.

2 Related Work

Network storage systems have engendered a vast literature, much of it focused on the design and performance of traditional client server systems such as NFS, AFS, CIFS, and WAFL [6, 7, 8, 25]. Recently, a range of efforts has considered other structures, including those based on peer-to-peer storage [16] among distributed sets of untrusted servers [12, 13] which have indirectly informed subsequent cloud-based designs.

Cloud storage is a newer topic, driven by the availability of commodity services from Amazon's S3 and other providers. The elastic nature of cloud storage is reminiscent of the motivation for the Plan 9 write-once file systems [19, 20], although cloud communication overheads and monetary costs argue against a block interface and no storage reclamation. Perhaps the closest academic work to our own is SafeStore [11], which stripes erasure-coded data objects across multiple storage providers, ultimately exploring access via an NFS interface. However, SafeStore is focused clearly on availability, rather than performance or cost, and thus its design decisions are quite different. A similar, albeit more complex system, is DepSky [2], which also focuses strongly on availability, proposing a "cloud of clouds" model to replicate across providers.

At a more abstract level, Chen and Sion create an economic framework for evaluating cloud storage costs and conclude that the computational costs of the cryptographic operations needed to ensure privacy can overwhelm other economic benefits [3]. However, this work predates Intel's AES-NI architecture extension which significantly accelerates data encryption operations.

There have also been a range of non-academic attempts to provide traditional file system interfaces for the key-value storage systems offered by services like Amazon's S3. Most of these install new per-client file system drivers. Exemplars include s3fs [22], which tries to map the file system directly on to S3's storage model (which both changes file system semantics, but also can dramatically increase costs) and ElasticDrive [5], which exports a block-level interface (potentially discarding optimizations that use file-level knowledge such as prefetching).

However, the systems closest to our own are "cloud storage gateways", a new class of storage server that has emerged in the last few years (contemporaneous with our effort). These systems, exemplified by companies such as Nasuni, Cirtas, TwinStrata, StorSimple and Panzura, provide caching network file system proxies (or "gateways") that are, at least on the surface, very similar to our design. Pricing schedules for these systems generally reflect a 2× premium over raw cloud storage costs. While few details of these systems are public, in general they validate the design point we have chosen.

Of commercial cloud storage gateways, Nasuni [17] is perhaps most similar to BlueSky. Nasuni provides a "virtual NAS appliance" (or "filer"), software packaged as a virtual machine which the customer runs on their own hardware—this is very much like the BlueSky proxy software that we build. The Nasuni filer acts as a cache and writes data durably to the cloud. Because Nasuni does not publish implementation details it is not possible to know precisely how similar Nasuni is to BlueSky, though there are some external differences. In terms of cost, Nasuni charges a price based simply on the quantity of disk space consumed (around $0.30/GB/month, depending on the cloud provider)—and not at all a function of data transferred or operations performed. Presumably, Nasuni optimizes their system to reduce the network and per-operation overheads—otherwise those would eat into their profits—but the details of how they do so are unclear, other than by employing caching.

Cirtas [4] builds a cloud gateway as well but sells it in appliance form: Cirtas's Bluejet is a rack-mounted computer which integrates software to cache file system data with storage hardware in a single package. Cirtas thus has a higher up-front cost than Nasuni's product, but is easier to deploy. Panzura [18] provides yet another CIFS/NFS gateway to cloud storage. Unlike BlueSky and the others, Panzura allows multiple customer sites to each run a cloud gateway. Each of these gateways accesses the same underlying file system, so Panzura is particularly appropriate for teams sharing data over a wide area. But again, implementation details are not provided.

TwinStrata [29] and StorSimple [28] implement gateways that present a block-level storage interface, like ElasticDrive, and thus lose many potential file system-level optimizations as well.

In some respects BlueSky acts like a local storage server that backs up data to the cloud—a local NFS server combined with Mozy [15], Cumulus [30], or similar software could provide similar functionality. How-

ever, such backup tools may not support a high backup frequency (ensuring data reaches the cloud quickly) and efficient random access to files in the cloud. Further, they treat the local data (rather than the cloud copy) as authoritative, preventing the local server from caching just a subset of the files.

3 Architecture

BlueSky provides service to clients in an enterprise using a transparent proxy-based architecture that stores data persistently on cloud storage providers (Figure 1). The enterprise setting we specifically consider consists of a single proxy cache colocated with enterprise clients, with a relatively high-latency yet high-bandwidth link to cloud storage, with typical office and engineering request workloads to files totaling tens of terabytes. This section discusses the role of the proxy and cloud provider components, as well as the security model supported by BlueSky. Sections 4 and 5 then describe the layout and operation of the BlueSky file system and the BlueSky proxy, respectively.

Cloud storage acts much like another layer in the storage hierarchy. However, it presents new design considerations that, combined, make it distinct from other layers and strongly influence its use as a file service. The high latency to the cloud necessitates aggressive caching close to the enterprise. On the other hand, cloud storage has elastic capacity and provides operation service times independent of spatial locality, thus greatly easing free space management and data layout. Cloud storage interfaces often only support writing complete objects in an operation, preventing the efficient update of just a portion of a stored object. This constraint motivates an append rather than an overwrite model for storing data.

Monetary cost also becomes an explicit metric of optimization: cloud storage capacity might be elastic, but still needs to be parsimoniously managed to minimize storage costs over time [30]. With an append model of storage, garbage collection becomes a necessity. Providers also charge a small cost for each operation. Although slight, costs are sufficiently high to motivate aggregating small objects (metadata and small files) into larger units when writing data. Finally, outsourcing data storage makes security a primary consideration.

3.1 Local Proxy

The central component of BlueSky is a proxy situated between clients and cloud providers. The proxy communicates with clients in an enterprise using a standard network file system protocol, and communicates with cloud providers using a cloud storage protocol. Our prototype supports both the NFS (version 3) and CIFS protocols for clients, and the RESTful protocols for the Amazon S3 and Windows Azure cloud services. Ideally, the proxy

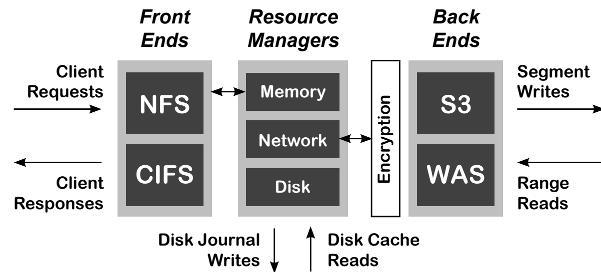

Figure 1: BlueSky architecture.

runs in the same enterprise network as the clients to minimize latency to them. The proxy caches data locally and manages sharing of data among clients without requiring an expensive round-trip to the cloud.

Clients do not require modification since they continue to use standard file-sharing protocols. They mount BlueSky file systems exported by the proxy just as if they were exported from an NFS or CIFS server. Further, the same BlueSky file system can be mounted by any type of client with shared semantics equivalent to Samba.

As described in more detail later, BlueSky lowers cost and improves performance by adopting a log-structured data layout for the file system stored on the cloud provider. A cleaner reclaims storage space by garbage-collecting old log segments which do not contain any live objects, and processing almost-empty segments by copying live data out of old segments into new segments.

As a write-back cache, the BlueSky proxy can fully satisfy client write requests with local network file system performance by writing to its local disk—as long as its cache capacity can absorb periods of write bursts as constrained by the bandwidth the proxy has to the cloud provider (Section 6.5). For read requests, the proxy can provide local performance to the extent that the proxy can cache the working set of the client read workload (Section 6.4).

3.2 Cloud Provider

So that BlueSky can potentially use any cloud provider for persistent storage service, it makes minimal assumptions of the provider; in our experiments, we use both Amazon S3 and the Windows Azure blob service. BlueSky requires only a basic interface supporting `get`, `put`, `list`, and `delete` operations. If the provider also supports a hosting service, BlueSky can co-locate the file system cleaner at the provider to reduce cost and improve cleaning performance.

3.3 Security

Security becomes a key concern with outsourcing critical functionality such as data storage. In designing BlueSky, our goal is to provide high assurances of data confiden-

tiality and integrity. The proxy encrypts all client data before sending it over the network, so the provider cannot read private data. Encryption is at the level of objects (inodes, file blocks, etc.) and not entire log segments. Data stored at the provider also includes integrity checks to detect any tampering by the storage provider.

However, some trust in the cloud provider is unavoidable, particularly for data availability. The provider can always delete or corrupt stored data, rendering it unavailable. These actions could be intentional—e.g., if the provider is malicious—or accidental, for instance due to insufficient redundancy in the face of correlated hardware failures from disasters. Ultimately, the best guard against such problems is through auditing and the use of multiple independent providers [2, 11]. BlueSky could readily incorporate such functionality, but doing so remains outside the scope of our current work.

A buggy or malicious storage provider could also serve stale data. Instead of returning the most recent data, it could return an old copy of a data object that nonetheless has a valid signature (because it was written by the client at an earlier time). By authenticating pointers between objects starting at the root, however, BlueSky prevents a provider from selectively rolling back file data. A provider can only roll back the entire file system to an earlier state, which customers will likely detect.

BlueSky can also take advantage of computation in the cloud for running the file system cleaner. As with storage, we do not want to completely trust the computational service, yet doing so provides a tension in the design. To maintain confidentiality, data encryption keys should not be available on cloud compute nodes. Yet, if cloud compute nodes are used for file system maintenance tasks, the compute nodes must be able to read and manipulate file system data structures. For BlueSky, we make the tradeoff of encrypting file data while leaving the metadata necessary for cleaning the file system unencrypted. As a result, storage providers can understand the layout of the file system, but the data remains confidential and the proxy can still validate its integrity.

In summary, BlueSky provides strong confidentiality and slightly weaker integrity guarantees (some data rollback attacks might be possible but are largely prevented), but must rely on the provider for availability.

4 BlueSky File System

This section describes the BlueSky file system layout. We present the object data structures maintained in the file system and their organization in a log-structured format. We also describe how BlueSky cleans the logs comprising the file system, and how the design conveniently lends itself to providing versioned backups of the data stored in the file system.

4.1 Object Types

BlueSky uses four types of objects for representing data and metadata in its log-structured file system [23] format: data blocks, inodes, inode maps, and checkpoints. These objects are aggregated into log segments for storage. Figure 2 illustrates their relationship in the layout of the file system. On top of this physical layout BlueSky provides standard POSIX file system semantics, including atomic renames and hard links.

Data blocks store file data. Files are broken apart into fixed-size blocks (except the last block may be short). BlueSky uses 32 KB blocks instead of typical disk file system sizes like 4 KB to reduce overhead: block pointers as well as extra header information impose a higher per-block overhead in BlueSky than in an on-disk file system. In the evaluations in Section 6, we show the cost and performance tradeoffs of this decision. Nothing fundamental, however, prevents BlueSky from using variable-size blocks optimized for the access patterns of each file, but we have not implemented this approach.

Inodes for all file types include basic metadata: ownership and access control, timestamps, etc. For regular files, inodes include a list of pointers to data blocks with the file contents. Directory entries are stored inline within the directory inode to reduce the overhead of path traversals. BlueSky does not use indirect blocks for locating file data—inodes directly contain pointers to all data blocks (easy to do since inodes are not fixed-size).

Inode maps list the locations in the log of the most recent version of each inode. Since inodes are not stored at fixed locations, inode maps provide the necessary level of indirection for locating inodes.

A checkpoint object determines the root of a file system snapshot. A checkpoint contains pointers to the locations of the current inode map objects. On initialization the proxy locates the most recent checkpoint by scanning backwards in the log, since the checkpoint is always one of the last objects written. Checkpoints are useful for maintaining file system integrity in the face of proxy failures, for decoupling cleaning and file service, and for providing versioned backup.

4.2 Cloud Log

For each file system, BlueSky maintains a separate log for each writer to the file system. Typically there are two: the proxy managing the file system on behalf of clients and a cleaner that garbage collects overwritten data. Each writer stores its log segments to a separate directory (different key prefix), so writers can make updates to the file system independently.

Each log consists of a number of log segments, and each log segment aggregates multiple objects together into an approximately fixed-size container for storage and transfer. In the current implementation segments are

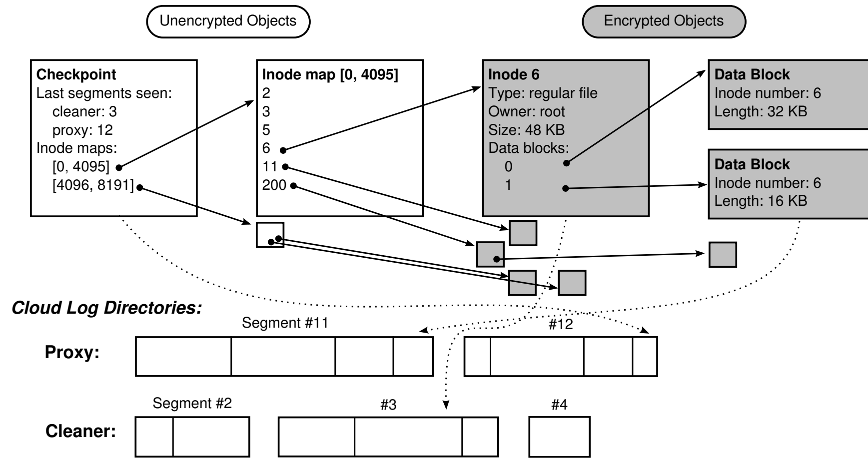

Figure 2: BlueSky filesystem layout. The top portion shows the logical organization. Object pointers are shown with solid arrows. Shaded objects are encrypted (but pointers are always unencrypted). The bottom of the figure illustrates how these log items are packed into segments stored in the cloud.

up to about 4 MB, large enough to avoid the overhead of dealing with many small objects. Though the storage interface requires that each log segment be written in a single operation, typically cloud providers allow partial reads of objects. As a result, BlueSky can read individual objects regardless of segment size. Section 6.6 quantifies the performance benefits of grouping data into segments and of selective reads, and Section 6.7 quantifies their cost benefits.

A monotonically-increasing *sequence number* identifies each log segment within a directory, and a byte *offset* identifies a specific object in the segment. Together, the triple $(directory, sequence\ number, offset)$ describes the physical location of each object. Object pointers also include the size of the object; while not required this hint allows BlueSky to quickly issue a read request for the exact bytes needed to fetch the object.

In support of BlueSky's security goals (Section 3.3), file system objects are individually encrypted (with AES) and protected with a keyed message authentication code (HMAC-SHA-256) by the proxy before uploading to the cloud service. Each object contains data with a mix of protections: some data is encrypted and authenticated, some data is authenticated plain-text, and some data is unauthenticated. The keys for encryption and authentication are not shared with the cloud, though we assume that customers keep a safe backup of these keys for disaster recovery. Figure 3 summarizes the fields included in objects.

BlueSky generates a unique identifier (UID) for each object when the object is written into the log. The UID remains constant if an item is simply relocated to a new log position. An object can contain pointers to other objects—for example, an inode pointing to data blocks—and the pointer lists both the UID and the physical location. A cleaner in the cloud can relocate objects and update pointers with the new locations; as long as the UID in the pointer and the object match, the proxy can validate that the data has not been tampered with.

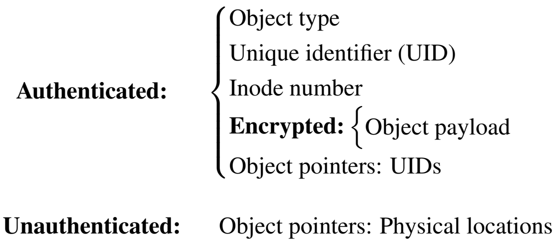

Figure 3: Data fields included in most objects.

4.3 Cleaner

As with any log-structured file system, BlueSky requires a file system cleaner to garbage collect data that has been overwritten. Unlike traditional disk-based systems, the elastic nature of cloud storage means that the file system can grow effectively unbounded. Thus, the cleaner is not necessary to make progress when writing out new data, only to reduce storage costs and defragment data for more efficient access.

We designed the BlueSky cleaner so that it can run either at the proxy or on a compute instance within the cloud provider where it has faster, cheaper access to the storage. For example, when running the cleaner in Amazon EC2 and accessing storage in S3, Amazon does not charge for data transfers (though it still charges for operations). A cleaner running in the cloud does not need to be fully trusted—it will need permission to read and write cloud storage, but does not require the file system encryption and authentication keys.

The cleaner runs online with no synchronous interactions with the proxy: clients can continue to access and modify the file system even while the cleaner is running. Conflicting updates to the same objects are later merged by the proxy, as described in Section 5.3.

4.4 Backups

The log-structured design allows BlueSky to integrate file system snapshots for backup purposes easily. In fact, so long as a cleaner is never run, any checkpoint record ever written to the cloud can be used to reconstruct the state of the file system at that point in time. Though not implemented in our prototype, the cleaner or a snapshot tool could record a list of checkpoints to retain and protect all required log segments from deletion. Those segments could also be archived elsewhere for safekeeping.

4.5 Multi-Proxy Access

In the current BlueSky implementation only a single proxy can write to the file system, along with the cleaner which can run in parallel. It would be desirable to have multiple proxies reading from and writing to the same BlueSky file system at the same time—either from a single site, to increase capacity and throughput, or from multiple sites, to optimize latency for geographically-distributed clients.

The support for multiple file system logs in BlueSky should make it easier to add support for multiple concurrent proxies. Two approaches are possible. Similar to Ivy [16], the proxies could be unsynchronized, offering loose consistency guarantees and assuming only a single site updates a file most of the time. When conflicting updates occur in the uncommon case, the system would present the user with multiple file versions to reconcile.

A second approach is to provide stronger consistency by serializing concurrent access to files from multiple proxies. This approach adds the complexity of some type of distributed lock manager to the system. Since cloud storage itself does not provide the necessary locking semantics, a lock manager would either need to run on a cloud compute node or on the proxies (ideally, distributed across the proxies for fault tolerance).

Exploring either option remains future work.

5 BlueSky Proxy

This section describes the design and implementation of the BlueSky proxy, including how it caches data in memory and on disk, manages its network connections to the cloud, and indirectly cooperates with the cleaner.

5.1 Cache Management

The proxy uses its local disk storage to implement a write-back cache. The proxy logs file system write requests from clients (both data and metadata) to a journal on local disk, and ensures that data is safely on disk before telling clients that data is committed. Writes are sent to the cloud asynchronously. Physically, the journal is broken apart into sequentially-numbered files on disk (journal segments) of a few megabytes each.

This write-back caching does mean that in the event of a catastrophic failure of the proxy—if the proxy's storage is lost—that some data may not have been written to the cloud and will be lost. If the local storage is intact no data will be lost; the proxy will replay the changes recorded in the journal. Periodically, the proxy snapshots the file system state, collects new file system objects and any inode map updates into one or more log segments, and uploads those log segments to cloud storage. Our prototype proxy implementation does not currently perform deduplication, and we leave exploring the tradeoffs of such an optimization for future work.

There are tradeoffs in choosing how quickly to flush data to the cloud. Writing data to the cloud quickly minimizes the window for data loss. However, a longer timeout has advantages as well: it enables larger log segment sizes, and it allows overlapping writes to be combined. In the extreme case of short-lived temporary files, no data need be uploaded to the cloud. Currently the BlueSky proxy commits data as frequently as once every five seconds. BlueSky does not start writing a new checkpoint until the previous one completes, so under a heavy write load checkpoints may commit less frequently.

The proxy keeps a cache on disk to satisfy many read requests without going to the cloud; this cache consists of old journal segments and log segments downloaded from cloud storage. Journal and log segments are discarded from the cache using an LRU policy, except that journal segments not yet committed to the cloud are kept pinned in the cache. At most half of the disk cache can be pinned in this way. The proxy sends HTTP byte-range requests to decrease latency and cost when only part of a log segment is needed. It stores partially-downloaded segments as sparse files in the cache.

5.2 Connection Management

The BlueSky storage backends reuse HTTP connections when sending and receiving data from the cloud; the CURL library handles the details of this connection pooling. Separate threads perform each upload or download. BlueSky limits uploads to no more than 32 segments concurrently, to limit contention among TCP sessions and to limit memory usage in the proxy (it buffers each segment entirely in memory before sending).

5.3 Merging System State

As discussed in Section 4.3, the proxy and the cleaner operate independently of each other. When the cleaner runs, it starts from the most recent checkpoint written by

```
merge_inode(ino_p, ino_c):
  if ino_p.id = ino_c.id:
    return ino_c    // No conflicting changes
  // Start with proxy version and merge cleaner changes
  ino_m ← ino_p; ino_m.id ← fresh_uuid(); updated ← false
  for i in [0 ... num_blocks(ino_p) − 1]:
    b_p ← ino_p.blocks[i]; b_c ← ino_c.blocks[i]
    if b_c.id = b_p.id and b_c.loc ≠ b_p.loc:
      // Relocated data by cleaner is current
      ino_m.blocks.append(b_c); updated ← true
    else:   // Take proxy's version of data block
      ino_m.blocks.append(b_p)
  return (ino_m if updated else ino_p)
```

Figure 4: Pseudocode for the proxy algorithm that merges state for possibly divergent inodes. Subscripts $_p$ and $_c$ indicate state written by the proxy and cleaner, respectively; $_m$ is used for a candidate merged version.

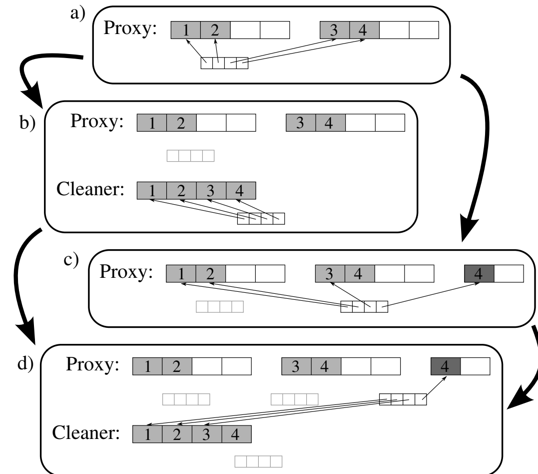

Figure 5: Example of concurrent updates by cleaner and proxy, and the resulting merged state.

the proxy. The cleaner only ever accesses data relative to this file system snapshot, even if the proxy writes additional updates to the cloud. As a result, the proxy and cleaner each may make updates to the same objects (e.g., inodes) in the file system. Since reconciling the updates requires unencrypted access to the objects, the proxy assumes responsibility for merging file system state.

When the cleaner finishes execution, it writes an updated checkpoint record to its log; this checkpoint record identifies the snapshot on which the cleaning was based. When the proxy sees a new checkpoint record from the cleaner, it begins merging updates made by the cleaner with its own updates.

BlueSky does not currently support the general case of merging file system state from many writers, and only supports the special case of merging updates from a single proxy and cleaner. This case is straightforward since only the proxy makes logical changes to the file system and the cleaner merely relocates data. In the worst case, if the proxy has difficulty merging changes by the cleaner it can simply discard the cleaner's changes.

The persistent UIDs for objects can optimize the check for whether merging is needed. If both the proxy and cleaner logs use the same UID for an object, the cleaner's version may be used. The UIDs will differ if the proxy has made any changes to the object, in which case the objects must be merged or the proxy's version used. For data blocks, the proxy's version is always used. For inodes, the proxy merges file data block-by-block according to the algorithm shown in Figure 4. The proxy can similarly use inode map objects directly if possible, or write merged maps if needed.

Figure 5 shows an example of concurrent updates by the cleaner and proxy. State (a) includes a file with four blocks, stored in two segments written by the proxy. At (b) the cleaner runs and relocates the data blocks. Concurrently, in (c) the proxy writes an update to the file, changing the contents of block 4. When the proxy merges state in (d), it accepts the relocated blocks 1–3 written by the cleaner but keeps the updated block 4. At this point, when the cleaner runs again it can garbage collect the two unused proxy segments.

5.4 Implementation

Our BlueSky prototype is implemented primarily in C, with small amounts of C++ and Python. The core BlueSky library, which implements the file system but not any of the front-ends, consists of 8500 lines of code (including comments and whitespace). BlueSky uses GLib for data structures and utility functions, libgcrypt for cryptographic primitives, and libs3 and libcurl for interaction with Amazon S3 and Windows Azure.

Our NFS server consists of another 3000 lines of code, not counting code entirely generated by the rpcgen RPC protocol compiler. The CIFS server builds on top of Samba 4, adding approximately 1800 lines of code in a new backend. These interfaces do not fully implement all file system features such as security and permissions handling, but are sufficient to evaluate the performance of the system. The prototype in-cloud file system cleaner is implemented in just 650 lines of portable Python code and does not depend on the BlueSky core library.

6 Evaluation

In this section we evaluate the BlueSky proxy prototype implementation. We explore performance from the proxy to the cloud, the effect of various design choices on both performance and cost, and how BlueSky performance varies as a function of its ability to cache client working sets for reads and absorb bursts of client writes.

6.1 Experimental Setup

We ran experiments on Dell PowerEdge R200 servers with 2.13 GHz Intel Xeon X3210 (quad-core) processors, a 7200 RPM 80 GB SATA hard drive, and gigabit network connectivity (internal and to the Internet). One machine, with 4 GB of RAM, is used as a load generator. The second machine, with 8 GB of RAM and an additional 1.5 TB 7200 RPM disk drive, acts as a standard file server or a BlueSky proxy. Both servers run Debian testing; the load generator machine is a 32-bit install (required for SPECsfs) while the proxy machine uses a 64-bit operating system. For comparison purposes we also ran a few tests against a commercial NAS filer in production use by our group. We focused our efforts on two providers: Amazon's Simple Storage Service (S3) [1] and Windows Azure storage [14]. For Amazon S3, we looked at both the standard US region (East Coast) as well as S3's West Coast (Northern California) region.

We use the SPECsfs2008 [27] benchmark in many of our performance evaluations. SPECsfs can generate both NFSv3 and CIFS workloads patterned after real-world traces. In these experiments, SPECsfs subjects the server to increasing loads (measured in operations per second) while simultaneously increasing the size of the working set of files accessed. Our use of SPECsfs for research purposes does not follow all rules for fully-compliant benchmark results, but should allow for relative comparisons. System load on the load generator machine remains low, and the load generator is not the bottleneck.

In several of the benchmarks, the load generator machine mounts the BlueSky file system with the standard Linux NFS client. In Section 6.4, we use a synthetic load generator which directly generates NFS read requests (bypassing the kernel NFS client) for better control.

6.2 Cloud Provider Bandwidth

To understand the performance bounds on any implementation and to guide our specific design, we measured the performance our proxy is able to achieve writing data to Amazon S3. Figure 6 shows that the BlueSky proxy has the potential to fully utilize its gigabit link to S3 if it uses large request sizes and parallel TCP connections. The graph shows the total rate at which the proxy could upload data to S3 for a variety of request sizes and number of parallel connections. Network round-trip time from the proxy to the standard S3 region, shown in the graph, is around 30 ms. We do not pipeline requests—we wait for confirmation for each object on a connection before sending another one—so each connection is mostly idle when uploading small objects. Larger objects better utilize the network, but objects of one to a few megabytes are sufficient to capture most gains. A single connection utilizes only a fraction of the total bandwidth, so to fully make use of the network we need multiple parallel

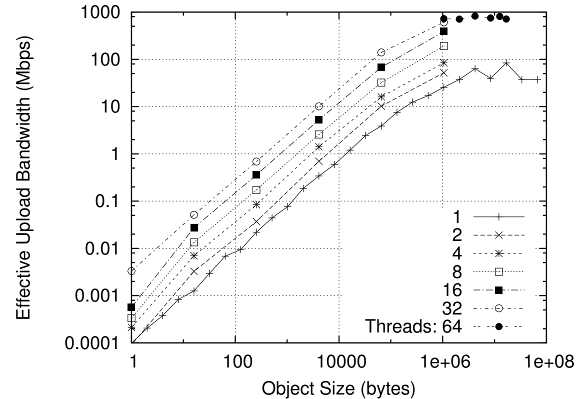

Figure 6: Measured aggregate upload performance to Amazon S3, as a function of the size of the objects uploaded (x-axis) and number of parallel connections made (various curves). A gigabit network link is available. Full use of the link requires parallel uploads of large objects.

TCP connections. These measurements helped inform the choice of 4 MB log segments (Section 4.1) and a pool size of 32 connections (Section 5.2).

The S3 US-West data center is closer to our proxy location and has a correspondingly lower measured round-trip time of 12 ms. The round-trip time to Azure from our location was substantially higher, around 85 ms. Yet network bandwidth was not a bottleneck in either case, with the achievable bandwidth again approaching 1 Gbps. In most benchmarks, we use the Amazon US-West region as the default cloud storage service.

6.3 Impact of Cloud Latency

To underscore the impact latency can have on file system performance, we first run a simple, time-honored benchmark of unpacking and compiling a kernel source tree. We measure the time for three steps: (1) extract the sources for Linux 2.6.37, which consist of roughly 400 MB in 35,000 files (a write-only workload); (2) checksum the contents of all files in the extracted sources (a read-only workload); (3) build an i386 kernel using the default configuration and the `-j4` flag for up to four parallel compiles (a mixed read/write workload). For a range of comparisons, we repeat this experiment on a number of system configurations. In all cases with a remote file server, we flushed the client's cache by unmounting the file system in between steps.

Table 1 shows the timing results of the benchmark steps for the various system configurations. Recall that the network links client↔proxy and proxy↔S3 are both 1 Gbps—the only difference is latency (12 ms from the proxy to BlueSky/S3-West and 30 ms to BlueSky/S3-East). Using a network file system, even locally, adds considerably to the execution time of the benchmark

	Unpack	*Check*	*Compile*
Local file system			
warm client cache	0:30	0:02	3:05
cold client cache		0:27	
Local NFS server			
warm server cache	10:50	0:26	4:23
cold server cache		0:49	
Commercial NAS filer			
warm cache	2:18	3:16	4:32
NFS server in EC2			
warm server cache	65:39	26:26	74:11
BlueSky/S3-West			
warm proxy cache	5:10	0:33	5:50
cold proxy cache		26:12	7:10
full segment		1:49	6:45
BlueSky/S3-East			
warm proxy	5:08	0:35	5:53
cold proxy cache		57:26	8:35
full segment		3:50	8:07

Table 1: Kernel compilation benchmark times for various file server configurations. Steps are (1) unpack sources, (2) checksum sources, (3) build kernel. Times are given in minutes:seconds. Cache flushing and prefetching are only relevant in steps (2) and (3).

compared to a local disk. However, running an NFS server in EC2 compared to running it locally increases execution times by a factor of 6–30× due to the high latency between the client and server and a workload with operations on many small files. In our experiments we use a local Linux NFS server as a baseline. Our commercial NAS filer does give better write performance than a Linux NFS server, likely due in part to better hardware and an NVRAM write cache. Enterprises replacing such filers with BlueSky on generic rack servers would therefore experience a drop in write performance.

The substantial impact latency can have on workload performance motivates the need for a proxy architecture. Since clients interact with the BlueSky proxy with low latency, BlueSky with a warm disk cache is able to achieve performance similar to a local NFS server. (In this case, BlueSky performs slightly better than NFS because its log-structured design is better-optimized for some write-heavy workloads; however, we consider this difference incidental.) With a cold cache, it has to read small files from S3, incurring the latency penalty of reading from the cloud. Ancillary prefetching from fetching full 4 MB log segments when a client requests data in any part of the segment greatly improves performance, in part because this particular benchmark has substantial locality; later on we will see that, in workloads with little locality, full segment fetches hurt performance. However, execution times are still multiples of BlueSky with a warm cache. The differences in latencies between S3-West and S3-East for the cold cache and full segment cases again underscore the sensitivity to cloud latency.

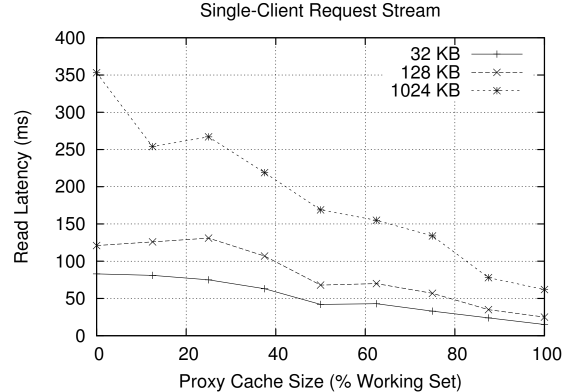

Figure 7: Read latency as a function of working set captured by the proxy. Results are from a single run.

In summary, greatly masking the high latency to cloud storage—even with high-bandwidth connectivity to the storage service—requires a local proxy to minimize latency to clients, while fully masking high cloud latency further requires an effective proxy cache.

6.4 Caching the Working Set

The BlueSky proxy can mask the high latency overhead of accessing data on a cloud service by caching data close to clients. For what kinds of file systems can such a proxy be an effective cache? Ideally, the proxy needs to cache the working set across all clients using the file system to maximize the number of requests that the proxy can satisfy locally. Although a number of factors can make generalizing difficult, previous studies have estimated that clients of a shared network file system typically have a combined working set that is roughly 10% of the entire file system in a day, and less at smaller time scales [24, 31]. For BlueSky to provide acceptable performance, it must have the capacity to hold this working set. As a rough back-of-the-envelope using this conservative daily estimate, a proxy with one commodity 3 TB disk of local storage could capture the daily working set for a 30 TB file system, and five such disks raises the file system size to 150 TB. Many enterprise storage needs fall well within this envelope, so a BlueSky proxy can comfortably capture working sets for such scenarios.

In practice, of course, workloads are dynamic. Even if proxy cache capacity is not an issue, clients shift their workloads over time and some fraction of the client workload to the proxy cannot be satisfied by the cache. To evaluate these cases, we use synthetic read and write workloads, and do so separately because they interact with the cache in different ways.

We start with read workloads. Reads that hit in the cache achieve local performance, while reads that miss

in the cache incur the full latency of accessing data in the cloud, stalling the clients accessing the data. The ratio of read hits and misses in the workload determines overall read performance, and fundamentally depends on how well the cache capacity is able to capture the file system working set across all clients in steady state.

We populate a BlueSky file system on S3 with 32 GB of data using 16 MB files.[1] We then generate a steady stream of fixed-size NFS read requests to random files through the BlueSky proxy. We vary the size of the proxy disk cache to represent different working set scenarios. In the best case, the capacity of the proxy cache is large enough to hold the entire working set: all read requests hit in the cache in steady state, minimizing latency. In the worst case, the cache capacity is zero, no part of the working set fits in the cache, and all requests go to the cloud service. In practice, a real workload falls in between these extremes. Since we make uniform random requests to any of the files, the working set is equivalent to the size of the entire file system.

Figure 7 shows that BlueSky with S3 provides good latency even when it is able to cache only 50% of the working set: with a local NFS latency of 21 ms for 32 KB requests, BlueSky is able to keep latency within 2× that value. Given that cache capacity is not an issue, this situation corresponds to clients dramatically changing the data they are accessing such that 50% of their requests are to new data objects not cached at the proxy. Larger requests take better advantage of bandwidth: 1024 KB requests are 32× larger than the 32 KB requests, but have latencies only 4× longer.

6.5 Absorbing Writes

The BlueSky proxy represents a classic write-back cache scenario in the context of a cache for a wide-area storage backend. In contrast to reads, the BlueSky proxy can absorb bursts of write traffic entirely with local performance since it implements a write-back cache. Two factors determine the proxy's ability to absorb write bursts: the capacity of the cache, which determines the instantaneous size of a burst the proxy can absorb; and the network bandwidth between the proxy and the cloud service, which determines the rate at which the proxy can drain the cache by writing back data. As long as the write workload from clients falls within these constraints, the BlueSky proxy can entirely mask the high latency to the cloud service for writes. However, if clients instantaneously burst more data than can fit in the cache, or if the steady-state write workload is higher than the bandwidth to the cloud, client writes start to experience delays that depend on the performance of the cloud service.

[1] For this and other experiments, we use relatively small file system sizes to keep the time for performing experiments manageable.

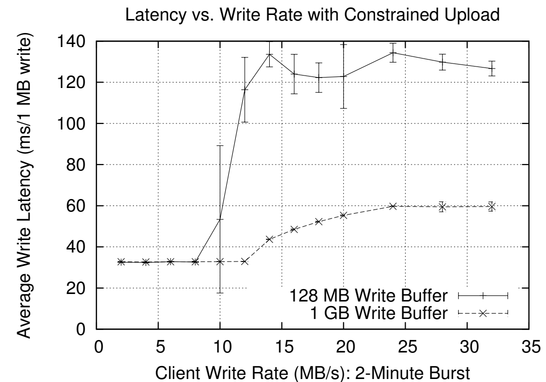

Figure 8: Write latencies when the proxy is uploading over a constrained (≈ 100 Mbps) uplink to S3 as a function of the write rate of the client and the size of the write cache to temporarily absorb writes.

We populate a BlueSky file system on S3 with 1 MB files and generate a steady stream of fixed-size 1 MB NFS write requests to random files in the file system. The client bursts writes at different rates for two minutes and then stops. So that we can overload the network between the BlueSky proxy and S3, we rate limit traffic to S3 at 100 Mbps while keeping the client↔proxy link unlimited at 1 Gbps. We start with a rate of write requests well below the traffic limit to S3, and then steadily increase the rate until the offered load is well above the limit.

Figure 8 shows the average latency of the 1 MB write requests as a function of offered load, with error bars showing standard deviation across three runs. At low write rates the latency is determined by the time to commit writes to the proxy's disk. The proxy can upload at up to about 12 MB/s to the cloud (due to the rate limiting), so beyond this point latency increases as the proxy must throttle writes by the client when the write buffer fills. With a 1 GB write-back cache the proxy can temporarily sustain write rates beyond the upload capacity. Over a 10 Mbps network (not shown), the write cache fills at correspondingly smaller client rates and latencies similarly quickly increase.

6.6 More Elaborate Workloads

Using the SPECsfs2008 benchmark we next examine the performance of BlueSky under more elaborate workload scenarios, both to subject BlueSky to more interesting workload mixes as well as to highlight the impact of different design decisions in BlueSky. We evaluate a number of different system configurations, including a native Linux nfsd in the local network (Local NFS) as well as BlueSky communicating with both Amazon S3's US-West region and Windows Azure's blob store. Unless otherwise noted, BlueSky evaluation results are for communication with Amazon S3. In addition to the base

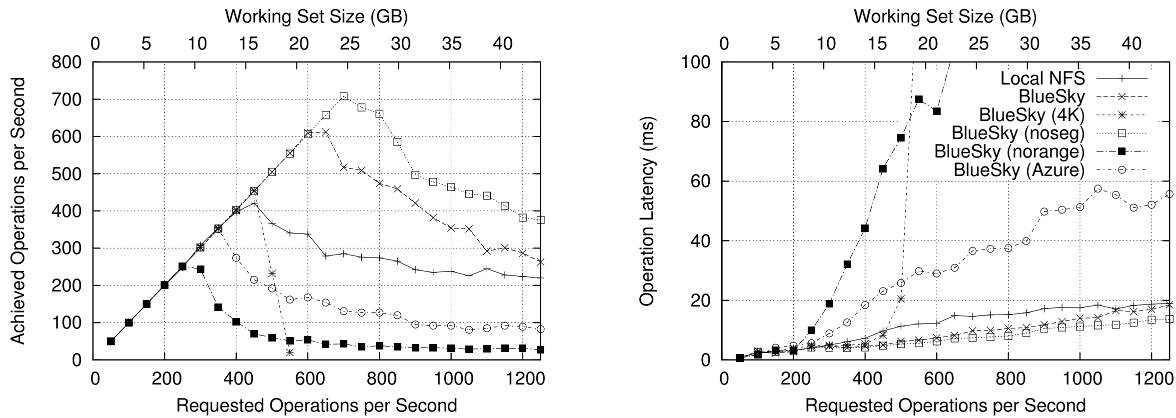

Figure 9: Comparison of various file server configurations subjected to the SPECsfs benchmark, with a low degree of parallelism (4 client processes). All BlueSky runs use cryptography, and most use Amazon US-West.

BlueSky configuration, we test a number of variants: disabling the log-structured design to store each object individually to the cloud (noseg), disabling range requests on reads so that full segments must be downloaded (norange), and using 4 KB file system blocks instead of the default 32 KB (4K). The "noseg" case is meant to allow a rough comparison with BlueSky had it been designed to store file system objects directly to the cloud (without entirely reimplementing it).

We run the SPECsfs benchmark in two different scenarios, modeling both low and high degrees of client parallelism. In the low-parallelism case, 4 client processes make requests to the server, each with at most 2 outstanding reads or writes. In the high-parallelism case, there are 16 client processes each making up to 8 reads or writes.

Figure 9 shows several SPECsfs runs under the low-parallelism case. In these experiments, the BlueSky proxy uses an 8 GB disk cache. The left graph shows the delivered throughput against the load offered by the load generator, and the right graph shows the corresponding average latency for the operations. At a low requested load, the file servers can easily keep up with the requests and so the achieved operations per second are equal to the requested load. As the server becomes saturated the achieved performance levels off and then decreases.

The solid curve corresponds to a local NFS server using one of the disks of the proxy machine for storage. This machine can sustain a rate of up to 420 operations/sec, at which point the disk is the performance bottleneck. The BlueSky server achieves a low latency—comparable to the local server case—at low loads since many operations hit in the proxy's cache and avoid wide-area network communication. At higher loads, performance degrades as the working set size increases. In write-heavy workloads, BlueSky incidentally performs better than the native Linux NFS server with local disk, since BlueSky commits operations to disk in a single journal and can make better use of disk bandwidth. Fundamentally, though, we consider using cloud storage successful as long as it provides performance commensurate with standard local network file systems.

BlueSky's aggregation of written data into log segments, and partial retrieval of data with byte-range requests, are important to achieving good performance and low cost with cloud storage providers. As discussed in Section 6.2, transferring data as larger objects is important for fully utilizing available bandwidth. As we show below, from a cost perspective larger objects are also better since small objects require more costly operations to store and retrieve an equal quantity of data.

In this experiment we also used Windows Azure as the cloud provider. Although Azure did not perform as well as S3, we attribute the difference primarily to the higher latency (85 ms RTT) to Azure from our proxy location (recall that we achieved equivalent maximum bandwidths to both services).

Figure 10 shows similar experiments but with a high degree of client parallelism. In these experiments, the proxy is configured with a 32 GB cache. To simulate the case in which cryptographic operations are better-accelerated, cryptography is disabled in most experiments but re-enabled in the "+crypto" experimental run. The "100 Mbps" test is identical to the base BlueSky experiment except that bandwidth to the cloud is constrained to 100 Mbps instead of 1 Gbps. Performance is comparable at first, but degrades somewhat and is more erratic under more intense workloads. Results in these experimental runs are similar to the low-parallelism case. The servers achieve a higher total throughput when there are more concurrent requests from clients. In the high-parallelism case, both BlueSky and the local NFS server provide comparable performance. Comparing cryptography enabled versus disabled, again there is very little difference: cryptographic operations are not a bottleneck.

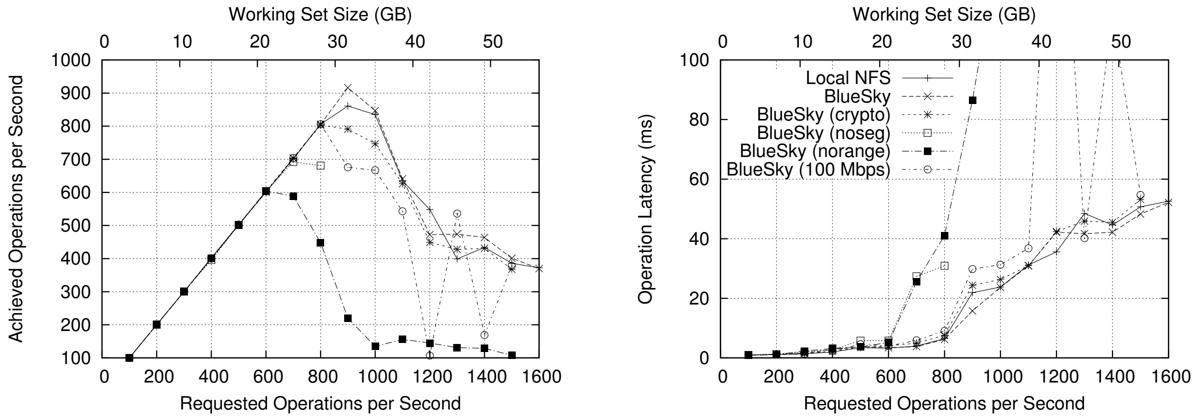

Figure 10: Comparison of various file server configurations subjected to the SPECsfs benchmark, with a high degree of parallelism (16 client processes). Most tests have cryptography disabled, but the "+crypto" test re-enables it.

	Down	*Op*	*Total*	*(Up)*
Baseline	$0.18	$0.09	$0.27	$0.56
4 KB blocks	0.09	0.07	0.16	0.47
Full segments	25.11	0.09	25.20	1.00
No segments	0.17	2.91	3.08	0.56

Table 2: Cost breakdown and comparison of various BlueSky configurations for using cloud storage. Costs are normalized to the cost per one million NFS operations in SPECsfs. Breakdowns include traffic costs for uploading data to S3 (Up), downloading data (Down), operation costs (Op), and their sum (Total). Amazon eliminated "Up" costs in mid-2011, but values using the old price are still shown for comparison.

6.7 Monetary Cost

Offloading file service to the cloud introduces monetary cost as another dimension for optimization. Figure 9 showed the relative performance of different variants of BlueSky using data from the low-parallelism SPECsfs benchmark runs. Table 2 shows the cost breakdown of each of the variants, normalized per SPECsfs operation (since the benchmark self-scales, different experiments have different numbers of operations). We use the September 2011 prices (in US Dollars) from Amazon S3 as the basis for the cost analysis: $0.14/GB stored per month, $0.12/GB transferred out, and $0.01 per 10,000 get or 1,000 put operations. S3 also offers cheaper price tiers for higher use, but we use the base prices as a worst case. Overall prices are similar for other providers.

Unlike performance, Table 2 shows that comparing by cost changes the relative ordering of the different system variants. Using 4 KB blocks had very poor performance, but using them has the lowest cost since they effectively transfer only data that clients request. The BlueSky baseline uses 32 KB blocks, requiring more data transfers and higher costs overall. If a client makes a 4 KB request, the proxy will download the full 32 KB block; many times downloading the full block will satisfy future client requests with spatial locality, but not always. Finally, the range request optimization is essential in reducing cost. When the proxy downloads an entire 4 MB segment when a client requests any data in it, the cost for downloading data increases by 150×. If providers did not support range requests, BlueSky would have to use smaller segments in its file system layout.

Although 4 KB blocks have the lowest cost, we argue that using 32 KB blocks has the best cost-performance tradeoff. The costs with 32 KB clocks are higher, but the performance of 4 KB blocks is far too low for a system that relies upon wide-area transfers

6.8 Cleaning

As with other file systems that do not overwrite in place, BlueSky must clean the file system to garbage collect overwritten data—although less to recover critical storage space, and more to save on the cost of storing unnecessary data at the cloud service. Recall that we designed the BlueSky cleaner to operate in one of two locations: running on the BlueSky proxy or on a compute instance in the cloud service. Cleaning in the cloud has compelling advantages: it is faster, does not consume proxy network bandwidth, and is cheaper since cloud services like S3 and Azure do not charge for local network traffic.

The overhead of cleaning fundamentally depends on the workload. The amount of data that needs to be read and written back depends on the rate at which existing data is overwritten and the fraction of live data in cleaned segments, and the time it takes to clean depends on both. Rather than hypothesize a range of workloads, we describe the results of a simple experiment to detail how the cleaner operates.

We populate a small BlueSky file system with 64 MB of data, split across 8 files. A client randomly writes, every few seconds, to a small portion (0.5 MB) of one of

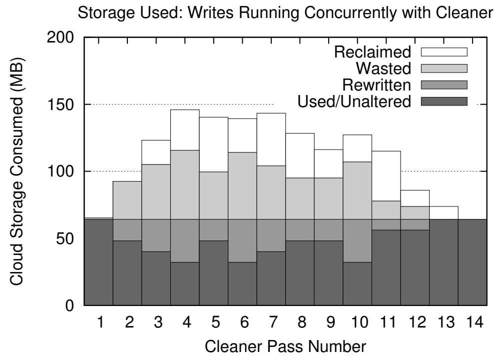

Figure 11: Storage space consumed during a write experiment running concurrently with the cleaner.

these files. Over the course of the experiment the client overwrites 64 MB of data. In parallel a cleaner runs to recover storage space and defragment file contents; the cleaner runs every 30 seconds, after the proxy incorporates changes made by the previous cleaner run. In addition to providing data about cleaner performance, this experiment validates the design that allows for safe concurrent execution of both the proxy and cleaner.

Figure 11 shows the storage consumed during this cleaner experiment; each set of stacked bars shows storage after a pass by the cleaner. At any point in time, only 64 MB of data is live in the file system, some of which (bottom dark bar) consists of data left alone by the cleaner and some of which (lighter gray bar) was rewritten by the cleaner. Some wasted space (lightest gray) cannot be immediately reclaimed; this space is either mixed useful data/garbage segments, or data whose relocation the proxy has yet to acknowledge. However, the cleaner deletes segments which it can establish the proxy no longer needs (white) to reclaim storage.

This workload causes the cleaner to write large amounts of data, because a small write to a file can cause the entire file to be rewritten to defragment the contents. Over the course of the experiment, even though the client only writes 64 MB of data the cleaner writes out an additional 224 MB of data. However, all these additional writes happen within the cloud where data transfers are free. The extra activity at the proxy, to merge updates written by the cleaner, adds only 750 KB in writes and 270 KB in reads.

Despite all the data being written out, the cleaner is able to reclaim space during experiment execution to keep the total space consumption bounded, and when the client write activity finishes at the end of the experiment the cleaner can repack the segment data to eliminate all remaining wasted space.

6.9 Client Protocols: NFS and CIFS

Finally, we use the SPECsfs benchmark to confirm that the performance of the BlueSky proxy is independent of the client protocol (NFS or CFS) that clients use. The experiments performed above use NFS for convenience, but the results hold for clients using CIFS as well.

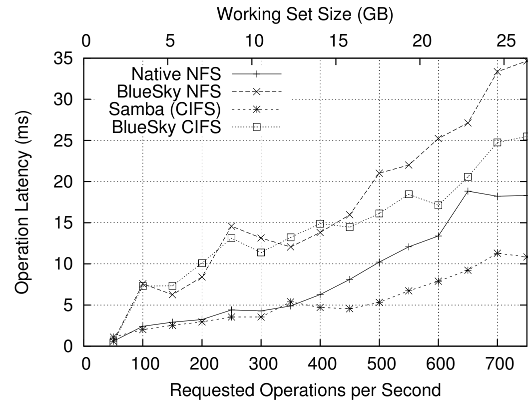

Figure 12: Latencies for read operations in SPECsfs as a function of aggregate operations per second (for all operations) and working set size.

Figure 12 shows the latency of the read operations in the benchmark as a function of aggregate operations per second (for all operations) and working set size. Because SPECsfs uses different operation mixes for its NFS and CIFS workloads, we focus on the latency of just the read operations for a common point of comparison. We show results for NFS and CIFS on the BlueSky proxy (Section 5.4) as well as standard implementations of both protocols (Linux NFS and Samba for CIFS, on which our implementation is based). For the BlueSky proxy and standard implementations, the performance of NFS and CIFS are broadly similar as the benchmark scales, and BlueSky mirrors any differences in the underlying standard implementations. Since SPECsfs uses a working set much larger than the BlueSky proxy cache capacity in this experiment, BlueSky has noticeably higher latencies than the standard implementations due to having to read data from cloud storage rather than local disk.

7 Conclusion

The promise of “the cloud” is that computation and storage will one day be seamlessly outsourced on an on-demand basis to massive data centers distributed around the globe, while individual clients will effectively become transient access portals. This model of the future (ironically similar to the old “big iron” mainframe model) may come to pass at some point, but today there are many hundreds of billions of dollars invested in the *last* disruptive computing model: client/server. Thus, in the interstitial years between now and a potential future built around cloud infrastructure, there will be a need to bridge the gap from one regime to the other.

In this paper, we have explored a solution to one such challenge: network file systems. Using a caching proxy

architecture we demonstrate that LAN-oriented workstation file system clients can be transparently served by cloud-based storage services with good performance for enterprise workloads. However, we show that exploiting the benefits of this arrangement requires that design choices (even low-level choices such as storage layout) are directly and carefully informed by the pricing models exported by cloud providers (this coupling ultimately favoring a log-structured layout with in-cloud cleaning).

8 Acknowledgments

We would like to thank our shepherd Ted Wong and the anonymous reviewers for their insightful feedback, and Brian Kantor and Cindy Moore for research computing support. This work was supported in part by the UCSD Center for Networked Systems. Vrable was further supported in part by a National Science Foundation Graduate Research Fellowship.

References

[1] Amazon Web Services. Amazon Simple Storage Service. `http://aws.amazon.com/s3/`.

[2] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. In *EuroSys 2011*, Apr. 2011.

[3] Y. Chen and R. Sion. To Cloud Or Not To Cloud? Musings On Costs and Viability. `http://www.cs.sunysb.edu/~sion/research/cloudc2010-draft.pdf`.

[4] Cirtas. Cirtas Bluejet Cloud Storage Controllers. `http://www.cirtas.com/`.

[5] Enomaly. ElasticDrive Distributed Remote Storage System. `http://www.elasticdrive.com/`.

[6] I. Heizer, P. Leach, and D. Perry. Common Internet File System Protocol (CIFS/1.0). `http://tools.ietf.org/html/draft-heizer-cifs-v1-spec-00`.

[7] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the Winter USENIX Technical Conference*, 1994.

[8] J. Howard, M. Kazar, S. Nichols, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, Feb. 1988.

[9] IDC. Global market pulse. `http://i.dell.com/sites/content/business/smb/sb360/en/Documents/0910-us-catalyst-2.pdf`.

[10] Jungle Disk. `http://www.jungledisk.com/`.

[11] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A Durable and Practical Storage System. In *Proceedings of the 2007 USENIX Annual Technical Conference*, June 2007.

[12] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[13] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage with Minimal Trust. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[14] Microsoft. Windows Azure. `http://www.microsoft.com/windowsazure/`.

[15] Mozy. `http://mozy.com/`.

[16] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th Conference on Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[17] Nasuni. Nasuni: The Gateway to Cloud Storage. `http://www.nasuni.com/`.

[18] Panzura. Panzura. `http://www.panzura.com/`.

[19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 From Bell Labs. *USENIX Computing Systems*, 8(3):221–254, Summer 1995.

[20] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[21] Rackspace. Rackspace Cloud. `http://www.rackspacecloud.com/`.

[22] R. Rizun. s3fs: FUSE-based file system backed by Amazon S3. `http://code.google.com/p/s3fs/wiki/FuseOverAmazon`.

[23] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[24] C. Ruemmler and J. Wilkes. A trace-driven analysis of disk working set sizes. Technical Report HPL-OSR-93-23, HP Labs, Apr. 1993.

[25] R. Sandberg, D. Goldberg, S. Kleirnan, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Technical Conference*, pages 119–130, 1985.

[26] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI), Apr. 2004. RFC 3720, `http://tools.ietf.org/html/rfc3720`.

[27] Standard Performance Evaluation Corporation. SPECsfs2008. `http://www.spec.org/sfs2008/`.

[28] StorSimple. StorSimple. `http://www.storsimple.com/`.

[29] TwinStrata. TwinStrata. `http://www.twinstrata.com/`.

[30] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem Backup to the Cloud. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2009.

[31] T. M. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[32] N. Zhu, J. Chen, and T.-C. Chiueh. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, Dec. 2005.