Knockoff: Cheap versions in the cloud

Xianzheng Dou, Peter M. Chen, and Jason Flinn
University of Michigan

Abstract

Cloud-based storage provides reliability and ease-of-management. Unfortunately, it can also incur significant costs for both storing and communicating data, even after using techniques such as chunk-based deduplication and delta compression. The current trend of providing access to past versions of data exacerbates both costs.

In this paper, we show that deterministic recomputation of data can substantially reduce the cost of cloud storage. Borrowing a well-known dualism from the fault-tolerance community, we note that any data can be equivalently represented by a log of the nondeterministic inputs needed to produce that data. We design a file system, called Knockoff, that selectively substitutes nondeterministic inputs for file data to reduce communication and storage costs. Knockoff compresses both data and computation logs: it uses chunk-based deduplication for file data and delta compression for logs of nondeterminism. In two studies, Knockoff reduces the average cost of sending files to the cloud without versioning by 21% and 24%; the relative benefit increases as versions are retained more frequently.

1 Introduction

Two trends in storage systems are conspiring to increase the cost of storing and retrieving data. First, due to compelling ease-of-management, cost-effectiveness, and reliability benefits, businesses and consumers are storing more of their data at cloud-based storage providers. However, writing and reading data from remote sites can incur significant costs for network communication.

Second, customers are increasingly expecting and depending on the ability to access multiple versions of their data. Local storage solutions such as Apple's Time Machine [3] retain multiple versions of users' data and make it easy to access this data. Cloud storage providers have followed suit; for example, Google Drive [18], Microsoft OneDrive [25], and DropBox [14] all store and allow users to access old versions of their files.

Past versions have many uses; e.g., recovery of lost or overwritten data, reproduction of the process by which data was created, auditing, and forensic troubleshooting. The benefit of versioning increases as more versions are retained. For instance, if versions are retained every time a file is closed, the user is usually guaranteed a snapshot of file data with each save operation or when the application terminates. However, many applications save data only on termination; in such cases, all intermediate data created during application usage are unavailable for recovery and analysis. Saving data on every file system modification produces more frequent checkpoints, but cannot recover transient state in memory that never is written to the file system and, importantly, does not capture modifications to memory-mapped files. In the extreme, a user should be able to reproduce any past state in the file system or in application memory, a property we call *eidetic* versioning.

The cost of versioning also depends on the frequency at which versions are retained. For instance, retaining a version on every file modification incurs greater storage cost than retaining a version on every close, and the client will consume more bandwidth by sending a greater number of versions to cloud storage. Versioning policies must balance these benefits and costs. Many current systems choose infrequent versioning as a result.

In this paper, we seek to substantially reduce the cost of communicating file data between clients and servers; we also seek to reduce the cost of keeping multiple versions of data. Our work reduces resource usage and user costs for existing versioning policies. It also enables finer-grained versioning, e.g. eidetic versioning, that is infeasible in current distributed storage architectures.

To accomplish these goals, we leverage an unconventional method for communicating and storing file data. In lieu of the actual file data, we selectively represent a file as a log of the nondeterminism needed to recom-

pute the data (e.g., system call results, thread scheduling, and external data read by a process). With such a log, a file server can deterministically replay the computation to recreate the data in the file. Representing state as a log of nondeterminism is well known in the fault-tolerance community [16]; however, logs of nondeterminism are often quite large, and applying this idea requires that the logs for a computation be smaller than the output files produced. To address this problem, we apply recent ideas for reducing the size of individual logs [13], and we also use delta compression to reduce the collective size of logs of similar executions.

Representing data as a log of nondeterminism leads to several benefits in a distributed file system. First, it substitutes (re-)computation for communication and storage, and this can reduce total cost because computation in cloud systems is less costly than communication and storage. Second, it can reduce the number of bytes sent over the network when the log of nondeterminism is smaller than the data produced by the recorded computation. For the same reason, it can reduce the number of bytes stored by the cloud storage provider. Finally, representing data as a log of nondeterminism can support a wider range of versioning frequencies than prior methods.

This paper describes the design and implementation of a distributed file system, Knockoff, that selectively replaces file data with a log of the nondeterminism needed to produce that data for both communication with cloud servers and storage in the cloud. Knockoff supports several frequencies of versioning: no versioning at all, version on file close, version on every write system call, and version on every store instruction (for mapped files).

The contributions of this paper are:

- We provide the first general-purpose solution for operation shipping in a distributed file system by leveraging deterministic record and replay.
- We show how compression can be applied to computation as well as to storage by using delta compression to reduce the size of the logs of nondeterminism that represent such computation.
- We quantify the costs and benefits of general-purpose operation shipping in a distributed file system over actual file system usage.

We evaluate Knockoff by performing a multi-user study for a software development scenario and a 20-day, single-user longitudinal study. Without versioning, Knockoff reduced the average cost of sending files to the cloud in these studies by 24% and 21%, respectively. The benefit of using Knockoff increases as versions are retained with greater frequency. The cost of this approach is the performance overhead of recording executions (7-8% in our evaluation) and a greater delay in retrieving past versions (up to 60 seconds for our default settings).

2 Background and related work

Knockoff is based on the principle that one can represent data generated by computation either by value or by the log of inputs needed to reproduce the computation. We call this the principle of equivalence (between values and computation); it has been observed and used in many settings; e.g., fault tolerance [16], state machine replication [37], data center storage management [20], and state synchronization [19].

The projects most related to ours use the principle of equivalence for the same purpose, namely to reduce communication overhead between clients and servers in a distributed file system. Lee et al. first applied this principle in the Coda File System [22, 23] and coined the term *operation shipping*. Clients log and send user operations (e.g., shell commands) to a server surrogate that replays the operations to regenerate the data. Chang et al. extend this idea to log and send user activity, such as keyboard and mouse inputs [10].

Although the basic idea of operation shipping is powerful, prior system logged and shipped very restricted types of nondeterminism and thus could not guarantee that the state received through the log matched the original state. Neither a log of shell commands nor a log of user activity are sufficient to reproduce the computation of general-purpose programs. Researchers recognized this shortcoming and mitigated it by supplementing the replayed computation with forward error correction and compensating actions, using hashes to detect remaining differences and revert back to value shipping. Unfortunately, the shift to multiprocessors and multithreaded programs means that many programs are nondeterministic in ways not handled in these prior systems. Further, because these prior systems handled a very limited set of nondeterministic inputs, they required identical environments on the recording and replaying side, which is unrealistic in many client-server settings.

Knockoff applies the same basic principle of equivalence, but it uses a comprehensive log of nondeterminism to provide equivalence for all race-free programs (and many programs with occasional data races). This enables Knockoff to use operation shipping in more settings, and it also makes possible the first realistic evaluation of operation shipping for such settings (earlier studies unrealistically assumed that programs were deterministic). Knockoff also applies operation shipping to versioning file systems. We find that the gains of operation shipping are larger when multiple versions are saved, and indispensable when versions are saved at eidetic granularity.

Adams et al. identify recomputation as a way to reduce storage [1], but do not implement or evaluate any system based on this observation. Other systems use recomputation to reduce storage in restricted environments

	Log entry	Values
1	open	rc=3
2	mmap	file=<id,version>
3	pthread_lock	
4	open	rc=4
5	read	rc=<size>, file=<id,version>
6	gettimeofday	rc=0, time=<timestamp>
7	open	rc=5
8	write	rc=<size>
9	pthread_unlock	

Figure 1: Sample log of nondeterminism

in which the computation is guaranteed to be deterministic. Nectar [20] applies this idea to DryadLINQ applications, which are both deterministic and functional. BAD-FS [6] uses re-computation in lieu of replicating data; users must specify explicit dependencies and the computation must be deterministic to produce the same data.

Besides reproducing data, logging has been used in file systems to track the provenance of files [29, 41] and guide when new versions should be saved [28]. More generally, redo logging [26] provides transactional properties in the presence of failures.

Many prior systems deduplicate file data to reduce communication and storage [12, 21, 31, 38, 40, 43]. LBFS [31] uses chunk-based deduplication in which Rabin fingerprinting divides a file into chunks, a hash value is computed for each chunk, and a client and server use the hash values to avoid communicating chunks already seen by the other party. Knockoff uses LBFS-style deduplication when transferring data by value.

Versioning file systems [24, 30, 36, 39, 44] retain past state at a specific granularity such as on every file close or on every modification. Cloud storage providers such as DropBox [14] and Google Drive [18] currently allow users to retain past versions of file data. Knockoff makes versioning file systems more efficient by reducing storage and computation costs. It also supports versioning at finer granularities than these prior systems.

3 Motivating example

We start with a motivating example that illustrates why a log of nondeterminism for an execution may require significantly less storage than the data produced by the execution. Consider a simple application that reads in a data file, computes a statistical transformation over that data, and writes a timestamped summary to an output file. The output data may be many megabytes in size. However, the program itself can be reproduced given a small log of determinism, as shown in Figure 1 (for clarity, the log has been simplified).

The log records the results of system calls (e.g., `open`) and synchronization operation (e.g., `pthread_lock`). The first entry in Figure 1 records the file descriptor chosen by the operating system during the original execution. Parameters to the open call do not need to be logged since they will be reproduced during a deterministic re-execution. The second entry records the mapping of the executable; replaying this entry will cause the exact version used during recording to be mapped to the same place in the replaying process address space. Lines 4 and 5 read data from the input file, line 6 records the original timestamp, and lines 7 and 8 write the transformation to the output file. Note that data read from the file system is not in the log since Knockoff can reproduce the desired version on demand. Also, the data written to the output file need not be logged since it will be reproduced exactly as a result of replaying the execution.

With compression, a log for this sample application can be only a few hundred bytes in size, as contrasted with the megabytes of data that the execution produces. The output data is reproduced by starting from the same initial state, re-executing the computation, and supplying values from the log for each nondeterministic operation.

4 Design considerations

Recent work on deterministic replay [2, 13, 15, 32, 35, 42] now makes it possible to use operation shipping to build a general-purpose distributed file system for realistic environments and workloads. Our goals are to build such a system, identify synergies between operation shipping and versioning file systems, and demonstrate how operation shipping can reduce communication and storage costs for realistic workloads.

4.1 Deterministic record and replay

To use operation shipping for realistic workloads and environments, we need a general-purpose deterministic record and replay system. The record/replay system should support unmodified applications and work for multithreaded programs. To work in realistic client/server configurations, the record/replay system should allow recorded executions to be replayed in environments that differ from the one on which they were recorded. Finally, to enable operation shipping to be used for some (but not all) processes, the system should record each application individually and allow each to be replayed individually on the server.

Knockoff uses the Arnold system [13], which meets these requirements. Arnold uses a modified Linux kernel to record the execution of Linux processes. It records all nondeterministic data that enters a process, including the results of system calls (such as user and network input), the timing of signals, and real-time clock queries. Because it supplies recorded values on replay rather than

re-executing system calls that interact with external dependencies, Arnold can trivially record an application on one computer and replay it on another. The only requirements are that both computers run the Arnold kernel and have the same processor architecture (x86).

Arnold enables deterministic replay of multi-threaded programs by recording all synchronization operations (e.g., `pthread_lock` and atomic hardware instructions). Arnold can detect programs with data races, but it does not guarantee that the replay of such programs will match their recorded execution. Arnold does guarantee that a replay is always repeatable (i.e., deterministic with respect to other replays), even for racy programs.

4.2 Files: values or operations?

Knockoff can represent a file in one of two ways: as normal file data (by value) or as a log of the nondeterminism needed to recreate the file (by operation). Which of these representations is more cost-effective depends on the characteristics of the program that generated the file data, as well as the relative costs of computation, communication, and storage. Files that are large and generated by programs that are mostly deterministic (e.g., photo-editing software) are best represented by operation. In contrast, files that are small and generated by programs that use a lot of nondeterministic data (e.g,. cryptographic key generation) are best represented by value.

At any time, Knockoff can use deterministic replay to convert a file that is represented by operation into a representation by value (but not vice versa). To do so, Knockoff loads and re-executes the original program, then feeds in the log of nondeterminism that was recorded in the original execution. Note that file data read by system calls from Knockoff are *not* included in the log. Instead, these log entries refer to the file and version that was read, and Arnold's replay system reads this data from Knockoff. Usually, application binaries, dynamic libraries, configuration files, and the like are stored in Knockoff, so the server replay sees the same application files as existed on the client during the original recording. If a binary or library is not stored in Knockoff, the file data are included by value in the log of nondeterminism and provided on replay. Replay of previously recorded applications may require retention of past file versions. Alternatively, we can regenerate these past versions through additional recursive replays of the applications that produced the data.

Whenever Knockoff represents a file by operation, it must first verify that Arnold's replay faithfully reconstructs the file data because Arnold does not guarantee that programs with data races replay exactly as recorded. Knockoff uses a SHA-512 hash for each file to verify that the replay correctly generated the original data. Because replay in Arnold is repeatable, a run that produces matching data in the first replay is guaranteed to produce matching data in all subsequent replays. If the replay does not produce matching data, Knockoff switches to representing the file by value.

Knockoff chooses between these two representations when it ships files between clients and servers and when it stores files on the server. To guide its choice, Knockoff measures the computation time used to create each file and the size of each file.

5 Implementation

Knockoff is a client-server distributed file system in which the server is hosted in the cloud. The server stores the current version of all files, and it optionally stores past versions of all files according to a user-selected versioning policy. Knockoff clients have a local disk cache that stores current and (optionally) past file versions.

Knockoff implements Coda-style weak file consistency [27]. Clients propagate file system updates asynchronously to the server. Clients register a callback with the server when they cache the current version of a file, and the server breaks the callback by sending a client a message when another client modifies the file.

Knockoff associates a version vector [33] with each file to identify specific versions and detect conflicting updates. Knockoff assigns clients a unique identifier; every time a client performs a system call that modifies a file (e.g., `write`), it increments an integer in the version vector associated with its identifier. Thus, every past version of a file has a unique version vector that can be used to name and retrieve that version. The server detects conflicting updates by comparing the version vector for each update and determining that neither one dominates the other. If a conflict occurs, the server retains both versions, and the user manually resolves the conflict.

Knockoff clients record almost all user-level process executions (excluding some servers such as the X server and sshd) and the kernel generates a log of nondeterminism for each such execution. Logs of nondeterminism are stored in a log cache on the client and may also be sent to the server and stored in a database there. The server has a replay engine that allows it to regenerate file data from such logs.

5.1 Versioning

Knockoff supports versioning policies on a per-file-system basis. Users select one of the following:

- **No versioning.** Knockoff retains only the current version of all files. For durability, a client sends a modified file to the server on `close`. After the first close, Knockoff waits up to 10 seconds to send the file modifications to the server (this delay allows coalescing of multiple updates that occur closely

together in time [27]). Upon receiving file modifications, the server overwrites the previous version of the file and breaks any callbacks held by other clients. The server retains multiple versions only in the case of conflicting updates.

- **Version on close.** Knockoff retains all past versions at close granularity; for past versions, Knockoff may store the actual data or the logs required to regenerate the data. On receiving a file modification, the server retains the previous version instead of overwriting it. Clients may ask for a version by specifying its unique version vector.
- **Version on write.** Knockoff retains all past versions at write granularity. Every system call that modifies a file creates a new version, and Knockoff can reproduce all such versions.
- **Eidetic.** Knockoff retains all past versions at instruction granularity. It can reproduce any computation or file data and determine the provenance of data via Arnold. The server stores all application logs. Clients may ask for a specific version of a file by specifying a version vector and an instruction count that specifies when to stop the replay (so as to recover a specific state for a mapped file).

5.2 Architecture

Figure 2 shows Knockoff's storage architecture. The client runs a FUSE[17] user-level file system daemon.

5.2.1 Clients

The Knockoff client stores file system data in four persistent caches to hide network latency. Whole file versions are stored in the *version cache*; this cache may hold multiple versions of the same file simultaneously. Each file in Knockoff is given a unique integer *fileid*, so a particular version of a file can be retrieved from the cache by specifying both a fileid and a version vector. The version cache tracks which versions it stores are the current version of a file. It sets callbacks for such entries; if the server breaks a callback (because another client has updated the file), the version is retained in the cache, but its designation as the current version is removed.

The *chunk cache* stores the chunks generated by chunk-based deduplication for each file in the version cache; thus the version cache contains only pointers to chunks in the chunk cache. Knockoff divides each entry in the database into chunks using the LBFS chunk-based deduplication algorithm [31] and calculates the SHA-512 hash of each such chunk. The chunk database is indexed by these hash values.

Directory data is stored in a separate Berkeley DB [7] *directory cache*. Knockoff clients implement an in-memory index over this data to speed up path lookups. The *log cache* stores logs of nondeterminism generated by recording application execution.

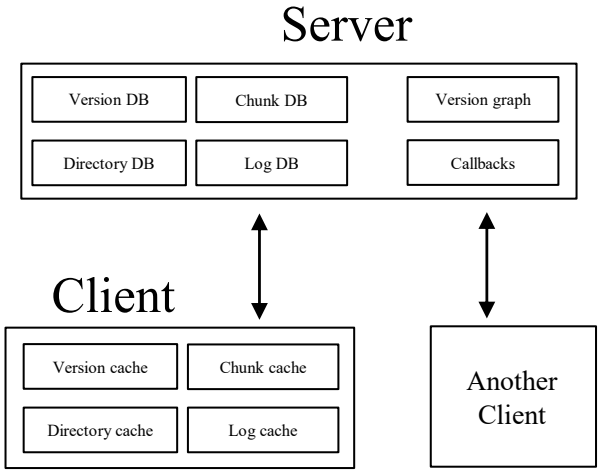


Figure 2: Architecture overview

All client caches are managed via LRU eviction, with preference given to retaining current versions over past versions. Chunks are removed from the chunk cache when they are no longer used by any version in the version cache. Modified values are pinned in the caches until they have been persisted to the server.

5.2.2 Server

The server maintains analogs of these client stores. The server's *version DB* stores the current version of every file and, depending on the the versioning policy, past file versions. The *chunk DB* stores chunk data for all files in the version DB, indexed by the chunk SHA-512 hash values. The *directory DB* stores all directory information for Knockoff, and the *log DB* stores logs of nondeterminism recorded by clients.

If the versioning policy is `eidetic`, the log DB stores every application log recorded by any Knockoff client. If the versioning policy is `version on write`, every past file version is either stored in the version DB, or the logs necessary to reproduce that version are stored in the log DB. If the versioning policy is `version on close`, this invariant holds only for versions that correspond to file closes. If the versioning policy is `no versioning`, only the current versions of file and directory data are stored.

The server also maintains callbacks set by clients for each file. If a client updates a file, the server uses these to determine which clients to notify of the update.

Finally, the server maintains the *version graph* which relates file versions with the computation that produced the data. Nodes in the graph are either recorded logs of nondeterminism (representing a particular execution) or file versions. Each edge represents a range of file data that was written by one recording and either read by another recording or part of a file version. An index allows Knockoff to quickly find a particular file version by fileid and version vector. If a version is not present in the ver-

sion DB, the version graph allows Knockoff to determine which logs need to be replayed to regenerate the data.

5.3 Writing data

Knockoff is designed to reduce cloud storage costs. A large part of these costs is communication. For example, AT&T [4] and Comcast [11] currently charge up to $10 for every 50 GB of data communication, whereas Amazon currently charges $0.052 per hour for an instance with 2 vCPUs and 4 GB of memory [5]. This means that Knockoff can reduce costs if it can use 1 second of cloud computation to save 76 KB of communication.

We first describe how Knockoff handles writes for its `no versioning` policy, and then generalize to other policies. Like Arnold, Knockoff records application execution in a *replay group* that logs the nondeterminism of related processes. When an application closes a file, the Knockoff client checks the file's version vector to determine whether it was modified. If so, Knockoff starts a 10-second timer that allows the application to make further modifications. When the timer expires, Knockoff starts a transaction in which all outstanding modifications to any file by that application are sent to the server. If the application terminates, Knockoff starts the transaction immediately.

To propagate modifications to the server, Knockoff first calculates the cost of sending and replaying the log of nondeterminism given a pre-defined cost of communication ($cost_{comm}$) and computation ($cost_{comp}$):

$$cost_{log} = size_{log} * cost_{comm} + time_{replay} * cost_{comp} \quad (1)$$

$size_{log}$ is determined by compressing the log of nondeterminism for the application that wrote the file and measuring its size directly. Because Knockoff does not currently support checkpointing, each log must be replayed from application start.

To estimate $time_{replay}$, Knockoff modifies Arnold to store the user CPU time consumed so far by the recorded application with each log entry that modifies file data. This is a very good estimate for the time needed to replay the log on the client [34]. To estimate server replay time, Arnold multiplies this value by a conversion factor to reflect the relative CPU speeds of the client and server.

Knockoff calculates the cost of sending file data as:

$$cost_{data} = size_{chunks} * cost_{comm} \quad (2)$$

Knockoff implements the chunk-based deduplication algorithm used by LBFS to reduce the cost of transmitting file data. It breaks all modified files into chunks, hashes each chunk, and sends the hashes to the server. The server responds with the set of hashes it has stored. $size_{chunks}$ is the size of any chunks unknown to the server that would need to be transmitted; Knockoff uses gzip compression to reduce bytes transmitted for such chunks.

If $cost_{log} < cost_{data}$, Knockoff sends the log to the server. The server makes a copy of the modified files and assigns them new version vectors. It then spawns a replay process that consumes the log and replays the application. When the replay process executes a system call that modifies a target file, it updates the new version in the cache directly. Once the replay reaches a file close operation, the new cache version is complete and marked as available to be read. The server deletes the old version for the `no versioning` policy. The replay process is paused at this point rather than terminated because the client may ship more log data to the server to regenerate additional file modifications made by the same application. The server only terminates a replay process when the client notifies it that the application that is being replayed has terminated.

In rare cases, more than one application may write to a file simultaneously. The Knockoff server replays these logs concurrently and ensures that writes are ordered correctly according to the version vectors.

Replay is guaranteed to produce the same data if the application being replayed is free of data races. Data-race freedom can be guaranteed for some programs (e.g., single-threaded ones) but not for complex applications. Knockoff therefore ships a SHA-512 hash of each modified file to the server with the log. The Knockoff server verifies this hash on close. If verification fails, it asks the client to ship the file data. Note that such races are rare with Arnold since the replay system itself acts as an efficient data-race detector [13].

If $cost_{data} < cost_{log}$, then Knockoff could reduce the cost of the current transaction by sending the unique chunks to the server. However, for long running applications, it may be the case that sending and replaying the log collected so far would help reduce the cost of future file modifications that have yet to be seen (because the cost of replaying from this point is less than replaying from the beginning of the program). Knockoff predicts this by looking at a history of $cost_{data}/cost_{log}$ ratios for the application. If sending logs has been historically beneficial and current application behavior is similar (the ratios differ by less than 40%) to past executions, it sends the log. Otherwise, it sends the unique data chunks.

For long running applications, Knockoff may use multiple transactions to send modified files to the server. If the client has previously chosen to ship the log to the server, $size_{log}$ is the portion of the log that excludes what has already been sent to the server, and $time_{replay}$ is the additional user CPU time consumed by the recorded application after the log prefix, scaled for the difference in client and server CPU speed.

Other versioning policies work similarly. For the `version on close` policy, Knockoff sends not only the current version of a modified file but also all versions that

existed at any file close during the period covered by the transaction. The cost of sending the log is identical to the `no versioning` case, but the previous versions may introduce new data chunks not yet seen by the server and thereby increase the cost of sending the file data.

For the `version on write` policy, the transaction includes all versions created by any system call that modified a file. Since chunk-based deduplication works better across larger modification sizes, Knockoff tries to coalesce modifications to the same file within a transaction as long as one modification does not overwrite data from a previous one. The transaction metadata describes how to reconstruct individual versions.

For the `eidetic` policy, Knockoff always sends the application log to the server since it is needed to reproduce past computations and mapped file state. The server usually updates the current version of files by replaying the log. However, if the computation cost of log replay is greater than the communication cost of fetching the modified file chunks from the server, the server asks the client for the file data instead of replaying the log.

5.4 Storing data

Knockoff may store file data on the server either by value (as normal file data) or by operation (as the log of nondeterminism required to recompute that data). If the log of nondeterminism is smaller than the file data it produces, then storing the file by operation saves space and money. However, storing files by operation delays future reads of that data, since Knockoff will need to replay the original computation that produced the data. In general, this implies that Knockoff should only store file data by operation if the data is very cold; i.e., if the probability of reading the data in the future is low.

Knockoff currently implements a simple policy to decide how to store data at the server. It always stores the current version of every file by value so that its read performance for current file data is the same as that of a traditional file system. Knockoff may store past versions by operation if the storage requirements for storing the data by log are less than those of storing the data by value. However, Knockoff also has a configuration parameter that sets a maximum *materialization delay*, which is the time to reconstruct any version stored by operation. The default materialization delay is 60 seconds.

For the `eidetic` policy, the materialization delay applies to all versions created via a file system call such as `write`. We could also apply this bound to intermediate file states for mapped files, but this would require us to implement checkpoints of application state so as to limit the time needed to produce intermediate states.

With the `eidetic` policy, Knockoff retains all logs of nondeterminism since they are needed to reproduce past computation and transient process state. This is sufficient to recompute any file version. However, the recomputation time is unbounded due to recursive dependencies. For instance, producing a past file version may require replaying a recorded computation. That computation may have read file data from Knockoff, so Knockoff must also reproduce those file versions via replay of other applications. This continues recursively until Knockoff encounters no more past versions to reproduce.

To limit the time to reproduce past versions, Knockoff selectively stores some past versions by value. It uses the server's version graph to decide which versions to store by value. The version graph shows the relationships between file versions and logs of nondeterminism. File versions and replay logs form the vertexes of the graph.

A version node in the graph contains a list of all the file byte ranges in that version that were written by distinct system calls. For each range, the node stores the log and the specific system call within the log that wrote the data. Replaying all such logs up to the specified system calls would be sufficient to recompute that particular version. However, recomputation is not necessary for byte ranges that already exist in the version DB. All byte ranges are present if the version represented by the node is itself in the version DB. Otherwise, a particular byte range may still be in the version DB because another version of the same file is stored by value and the range was not overwritten between the two versions.

Knockoff inserts an edge from a version node to a log node if the log contains a system call that wrote a byte range not in the version DB. Each edge has weight equal to the time to recompute all such byte ranges.

A log node contains similar information for each of its system calls that read data from a Knockoff file. For each read, it lists all file byte ranges that were written by distinct system calls; this contains the log and system call of the writing application. Replaying these logs would be sufficient to recompute all file system data read by the log node in question. Knockoff inserts an edge from one log node to another if the latter log wrote at least one byte range read by the former log that is not currently available in the version DB. As above, the weight of each edge is the predicted time to recompute all such byte ranges. If there is a cycle in the graph, two or more logs must be replayed concurrently to regenerate file data; Knockoff coalesces these cycles into a single node.

The time to recompute a version is given by the longest path rooted at the graph node for that version. Calculating the longest path for each version requires visiting each node at most once. If any path exceeds the specified materialization delay, Knockoff replays the latest log in the path, regenerates its file versions, and stores them by value. It repeats this process until no paths exceed the materialization delay. This greedy algorithm works well in our evaluation; if warranted in the future,

we could draw on more complex algorithms [8] to minimize storage costs while not exceeding the maximum materialization delay.

Currently, Knockoff recalculates the version graph and runs the above algorithm nightly. Note that file modifications and queries of past state between batch updates may have created new versions of past files in the version DB. These new versions are temporarily excluded from the batch computation. If Knockoff determines that they are not needed, they are removed from the version DB to save space. Otherwise, they are retained instead of recomputing them from logs of nondeterminism.

The `version on write` and `version on close` policies store data in similar fashion. The major difference is that these policies can discard logs to save storage space. Thus, for any log, if the size of the data produced by that log is less than the size of the log, Knockoff replays the log (if necessary) to recompute the data, then deletes the log. Discarded logs are removed from the version graph and the file versions produced by those logs are pinned in the version DB (they can never be deleted without violating the versioning policy since it is no longer possible to recompute their data).

5.5 Reading data

By default, any application that reads data from Knockoff receives the current version of the file. The client first checks its version cache to see if it has stored the current version of the file locally. If the version is present, Knockoff reads the requested data from the cache. It also appends a record to the reading application's log of nondeterminism that specifies the fileid, the version vector, and the logid and system call of the application that wrote the data it just read. The latter two values are obtained from Arnold's filemap [13].

If the version is not present, Knockoff fetches it from the server and caches it. Knockoff caches whole file versions, and clients fetch versions from the server by value. A client sends a request that specifies the fileid. The server responds with the current version vector and a list of hashes for each chunk comprising the file. The server also sends the filemap for the version. The client specifies which chunks it does not have cached, and the server sends that data to the client. The server sets a callback on the file. The client inserts the version into its version cache, marks it as the current version, and places the version's chunks into its chunk cache.

Applications may also read past versions of files by specifying a version vector. If a requested version is in the server's version DB, it is shipped by value as above. If it is not present, it must be recomputed by replaying one or more logs. We next describe this process.

In its version graph, the server maintains an index over all versions; this allows it to quickly find the particular version node being requested. The version node reveals the distinct byte ranges that were written by different system calls. If a range is in the version DB, it is used directly. Otherwise, the server must replay the log of the application that wrote the range to regenerate the data. For each such log, it determines the longest prefix that has to be replayed; this is the last system call in the log that wrote any range being read. Knockoff examines each such log prefix to determine if replaying the log requires file data that is not in the version cache. If so, it recursively visits the log(s) that wrote the needed data. Note that Knockoff's materialization delay bounds the amount of computation needed to produce any version. Knockoff then replays the visited logs to regenerate the desired version. It places this version in its version database and ships it to the client as described above.

5.6 Optimization: Log compression

While implementing Knockoff, we saw the effectiveness of chunk-based deduplication in reducing communication and storage costs. This led us to wonder: can we apply the same compression techniques to logs of nondeterminism that current file systems apply to file data?

Intuitively, log compression should be able to identify similar regions of nondeterministic data across executions of the same application. For example, application startup regions should be very similar because the application will open and load the same libraries, send similar messages to the X server to initialize windows, open similar configuration files, and so on.

We first attempted to apply chunk-based deduplication directly to log data. This worked reasonably well. However, after examining the logs we generated in more detail, we realized that the similarities between logs are often different from similarities between files. Similar files tend to have large contiguous chunks that are the same, whereas similar logs often lack such regions. Instead, most of the bytes within two log regions might be the same, but there exist in each region a smattering of values such as timestamps that differ. So, even very similar log chunks hash to different values.

Therefore, we turned to delta encoding. Knockoff first identifies a *reference log* that it expects to be similar to the current log. It then generates a binary patch via xdelta [45] that encodes the difference between the current log and the reference log. Given both the reference log and the patch, Knockoff can reconstruct the original values in the log.

When an application generates a log, the client and server identify a reference log. The client queries the log cache to find all prior logs for the same executable that it has stored locally. For each log, the log cache stores the arguments to the application, the size of the nondeterministic data, the running time of the application, and the

user-level CPU time. The client orders the cached logs by similarity across these metrics; if the application has not yet completed execution by the time the log is shipped, only the arguments are used to determine similarity since the other parameters are not yet known. Arguments are compared using cosine string similarity.

The client orders logs by similarity and sends the list to the server. The server responds with the most similar log that it has stored in its log DB. This step is omitted for the `eidetic` policy since the server stores all logs. The client then generates an xdelta patch and uses its size as $size_{log}$ in the algorithm described in Section 5.3.

When the server receives a compressed log, it stores it in compressed form in the log DB. It also adds a dependency on the reference log. Before a reference log can be pruned for the `version on close` or `version on write` policies, the server must first uncompress any log that depends on that log. The server uses the delta size when deciding whether to retain the log or the data in these policies. It currently does not take into account the cost of uncompressing logs when a reference log is purged because it assumes that logs can be recompressed effectively using different reference logs.

6 Evaluation

Our evaluation answers the following questions:

- How much does Knockoff reduce bandwidth usage compared to current cloud storage solutions?
- How much does Knockoff reduce communication and storage costs?
- What is Knockoff's performance overhead?
- How effective is log compression?

6.1 Experimental setup

All experiments for measuring communication and storage costs were run on two virtual machines (one for the client and one for the server). Both virtual machines were hosted on computers with a 4 core Intel i7-3770 CPU, 16GB memory, and two 7200 RPM hard drives. For accuracy, performance results were measured with a physical machine as the client with a 4 core Xeon E5620 processor, 6 GB memory, and a 7200 RPM hard drive. All platforms run the Ubuntu 12.04LTS Linux kernel.

Due to a lack of representative file system benchmarks that also include the computation to generate the data, we use two methods to evaluate Knockoff. First, we study users performing a software development task to measure how Knockoff benefits different people. Second, we measure Knockoff while an author of this paper runs the system on the author's primary computer for 20 days. This allows us to study the storage costs of multiple versions generated over a longer time period.

	20-day study	User study
Disk read (MB)	5473	2583
Disk write (MB)	6706	4339
File open count	261523	418594
Number of executions	3803	1146
Number of programs	75	63

Table 1: Workload characteristics

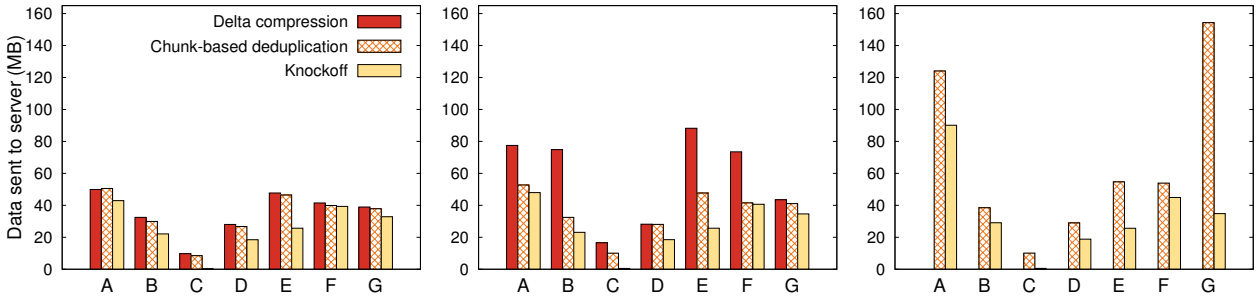
During these studies, we use Knockoff's `eidetic` policy, which allows us to regenerate all file system reads and writes by replaying Arnold logs. We use these logs to measure the bandwidth and storage costs of running Knockoff over the same workload with other policies.

We implement two baseline file systems for comparison. The first uses the LBFS algorithm for chunk-based deduplication to implement all versioning policies except `eidetic`; this is representative of current cloud storage solutions such as DropBox. The second uses delta compression to implement `no versioning` and `version on close`; this is representative of git [9] and other version control systems. Delta compression performed poorly for `version on write` because our implementation did not detect when bytes were inserted in the middle of a file; we therefore omit these results.

6.2 User study

We recruited 8 graduate students to study Knockoff for a software development workload. We asked them to write software to perform several simple tasks, e.g., converting a CSV file to a JSON file; each participant could spend up to an hour solving the problem. We did not dictate how the problem should be solved. Participants used various Linux utilities, text editors, IDEs, and programming languages. They used Web browsers to visit different Web sites such as Google and StackOverflow, as well as sites unrelated to the assignment (e.g., Facebook and CNN News). One of the 8 participants was unable to complete the programming assignment and quit right away. We show results for the 7 participants who attempted the tasks; 4 of these finished successfully within the hour. The second column of Table 1 shows aggregate characteristics of this recorded workload.

Figure 4 summarizes the results by aggregating the bytes sent to the server by Knockoff and the baseline file systems across all 7 users; this represents approximately 7 hours of software development activity. Although we are targeting versioning file systems, Knockoff is surprisingly effective in reducing bytes sent over the network for non-versioning file systems. Compared to chunk-based deduplication, Knockoff reduces communication by 24%. Compared to delta compression, it reduces communication by 32%. Note that the baselines are already very effective in reducing bandwidth; without compression, this workload requires 1.9 GB of commu-



(a) No versioning (b) Version on close (c) Version on write

Figure 3: Bytes sent to the server for each individual user study participant (A-G). We compare Knockoff with two baselines across all relevant versioning policies.

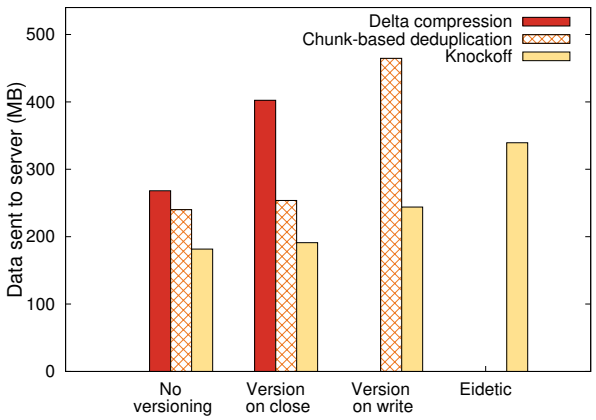


Figure 4: Total bytes sent to the server across all user study participants. We compare Knockoff with two baselines across all relevant versioning policies.

nication, so delta compression is achieving a 86% reduction in network bandwidth, and chunk-based deduplication is achieving a 87% reduction.

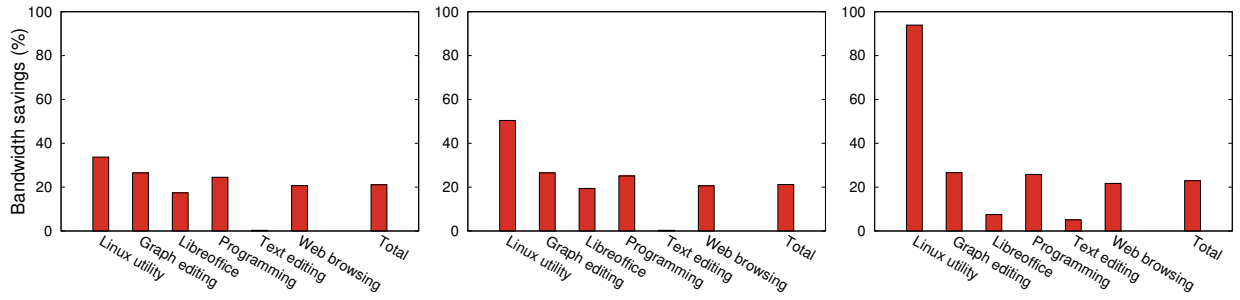
Results for `version on close` are similar to `no versioning` for two reasons: (1) the 10-second delay in transmitting data limits the amount of file closes that can be coalesced, and (2) file closes that occur within a few seconds of one another often save very similar data, so deduplication is very effective in reducing communication for both the baseline and Knockoff. For the `version on write` policy, Knockoff reduces bytes sent by 47% compared to chunk-based deduplication. Knockoff is very effective in reducing the additional cost of retaining fine-grained versions in this study; in fact, `version on write` with Knockoff use less bandwidth than `no versioning` with the baselines.

Figure 3 shows results for each individual study participant (labeled A-G in each graph). The most noticeable result is that the effectiveness of Knockoff varies tremendously across users. For participant C, Knockoff achieves a 97% reduction in bandwidth for the `no versioning` policy and a 95% reduction for the `version on write` policy compared to chunk-based deduplication. On the other hand, for participant F, the corresponding reductions are 2% and 17%. Participant C used more command line tools and repeated tasks than other participants. Participant F browsed Web sites more often. Unfortunately, Knockoff mispredicted whether to ship by log or value for browsing sessions and missed several opportunities for bandwidth reduction. Running a longer study might have allowed Knockoff to better model behavior and make better predictions for this user.

6.3 Bandwidth savings

To assess longer-term impacts of running Knockoff, one author ran Knockoff on his primary computer for 20 days. The usage was not continuous, as the author was simultaneously developing the system and fixing bugs. When in use, all user-level applications were recorded, and almost all data was stored in Knockoff. There were a few exceptions that included system directories, maintenance operations, and the execution of user-level daemons like the X server. Knockoff was initially populated by mirroring the current data in the computer's file system at the beginning of the trial; copying this data into Knockoff is excluded from our results. The first column of Table 1 shows aggregate characteristics of this recorded workload.

Figure 6 compares the bytes sent to the server by Knockoff with those sent by the baseline file systems. For the `no versioning` policy, Knockoff reduced bytes sent by 21% compared to chunk-based deduplication and by 39% compared to delta compression. Note that these compression techniques already reduce bytes sent by 84% and 79%, respectively, when compared to using no compression at all. For the `version on write` policy, Knockoff reduced bytes sent by 21% compared to chunk-based deduplication. In this experiment, Knockoff's implementation of fine-grained versioning policies is competitive with chunk-based deduplication without versioning, sending 21% more bytes to the cloud for `version on write` and 96% more for `eidetic`. This is a very



(a) No versioning (b) Version on close (c) Version on write

Figure 5: Relative reduction in bytes sent to the server for the 20-day study

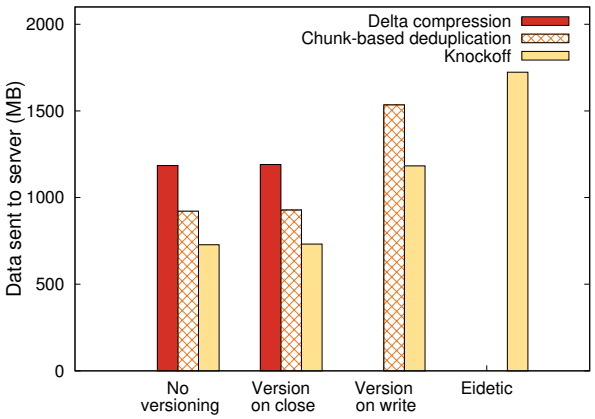


Figure 6: Bytes sent to the server for the 20-day study. We compare Knockoff with two baselines across all relevant versioning policies.

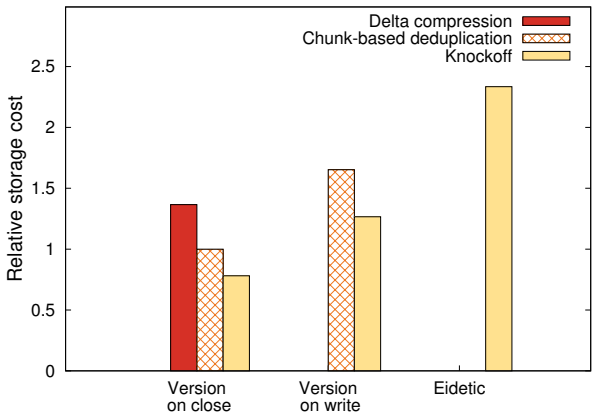


Figure 7: Relative storage costs for different versioning policies for the 20-day study.

encouraging result as it argues that retention of past state at fine granularity can be economically feasible.

To further explore these results, we manually classified the logs collected during the trial by application type. Figure 5 shows the reduction in bytes sent to the cloud for each type relative to chunk-based deduplication. Knockoff helps little for text editing because the log of nondeterminism is almost always larger than the data file produced by the editor. All other application types show robust reductions in bytes sent, with the savings being the greatest for Linux command line utilities.

6.4 Storage savings

We next examine how Knockoff impacts storage costs for the 20-day study. Storage costs typically depend on the amount of data stored; e.g., AWS currently charges $0.045 per GB-month [5]. Since Knockoff stores all current versions by value, we compare and report the amount of storage consumed by all file systems to store past versions. The Knockoff materialization delay limit for past versions is set to its default of 60 seconds.

Figure 7 shows the cost of storing versions of past state normalized to the chunk-based deduplication baseline. Compared to this baseline, Knockoff reduces storage utilization by 19% and 23% for the `version on close` and `version on write` policies, respectively.

Storage utilization increases as the version granularity gets smaller. However, with Knockoff, storing every write is only 21% more costly than the baseline system versioning on close, and eidetic storage is only 134% more costly. Thus, even versioning at eidetic granularity can be economically feasible if the storage system stores some past versions by operation rather than by value.

Figure 8 shows how changing the materialization delay impacts the relative storage cost. The values to the far right represent an infinite materialization delay, and thus show the minimum cost possible through varying this parameter.

6.5 Communication cost savings

We next measure the cost savings achieved by Knockoff. Sending file data to the server by operation reduces network usage, but it requires server-side computation to regenerate file data. We assess this tradeoff using current rates charged by popular Internet and cloud service providers. For network, we use a range of possible values. Popular broadband ISPs (AT&T [4] and Com-

	Price($ per GB)	Knockoff savings					
		No version		Version on close		Version on write	
		20-day study	User study	20-day study	User study	20-day study	User study
4G network	4.50	21.0%	21.8%	21.2%	21.7%	22.9%	46.3%
Expensive ISP	0.20	20.3%	18.4%	20.5%	18.5%	22.0%	43.3%
Cheap ISP	0.05	18.1%	13.8%	18.2%	14.5%	19.2%	34.9%
Hypothetical ISP	0.005	8.2%	4.9%	8.4%	5.8%	8.2%	11.5%

Table 2: Relative cost savings from using Knockoff for different versioning policies. We show costs for a typical 4G cellular network, an expensive current ISP, a cheap current ISP, and a hypothetical ISP that is an order of magnitude cheaper than the cheap current ISP.

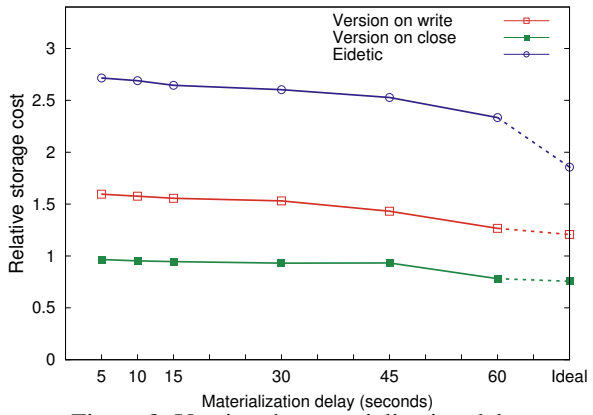


Figure 8: Varying the materialization delay

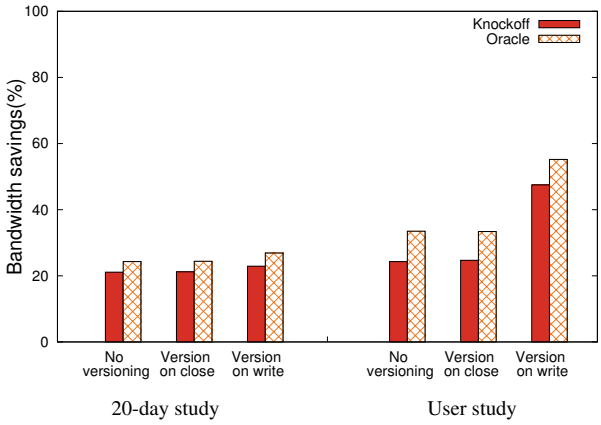


Figure 9: We compare Knockoff's actual bandwidth savings with those it could achieve with an oracle that perfectly predicts whether to ship by value or by operation.

cast [11]) currently charge a base ($50 per month) and an incremental charge ($10 for every 50 GB of data beyond a monthly data cap between 150GB and 1TB). Thus, we consider 2 price points: $0.05/GB and $0.20/GB. We also consider a typical 4G network, which has much higher cost ($4.50/GB or more), and a hypothetical cheap ISP ($0.005/GB) that is an order of magnitude lower than current providers. Cloud computation cost depends on the capabilities of the instance. AWS currently charges $0.052 per hour for an instance with 2 vCPUs and 4 GB of memory [5]. This is the cheapest instance sufficient to replay all of our logs, so we use this cost for our study.

Table 2 compares the monetary cost of sending data to the server for Knockoff and chunk-based deduplication (the best of the two baselines). For high network cost (4G), the cost savings of using Knockoff are essentially identical to the bandwidth savings. As network cost decreases, Knockoff's cost benefit diminishes. However, even for the hypothetical cheap ISP, Knockoff still achieves a healthy 4.9-11.5% reduction in dollar cost.

The reason why monetary cost savings aligns closely with network bandwidth savings for most network types is that the current tradeoff between communication and network costs is very favorable for operation shipping. Replaying applications is proportional to user-level CPU time because it eliminates user think-time and most I/O delays. When applications do not fit this profile, e.g., they have a lot of computation or large logs of nondeterminism, Knockoff usually ships the data by value.

6.6 Effectiveness of prediction

For long-running programs, Knockoff must predict whether it will be better to ship the output of that program by value or by operation. Mispredictions increase the bytes sent to the server. To measure this cost, we calculated the bytes that would be sent for our studies if an oracle were to perfectly predict which method Knockoff should use. As the results in Figure 9 show, better predictions could reduce network communication, but the potential improvement is not especially large.

6.7 Performance

We next examine Knockoff's performance overhead as perceived by its user. We measured this overhead by compiling libelf-0.8.9 with all source files, executables, and compilation outputs stored in Knockoff. We report the mean time to compile across 8 trials. Note that Knockoff sends data to the server asynchronously, and the server also replays logs asynchronously. As a baseline, we use a FUSE file system that simply forwards all file system operations to a local ext4 file system.

Figure 10 shows the results of our experiment. The first bar shows the baseline file system. The next bar

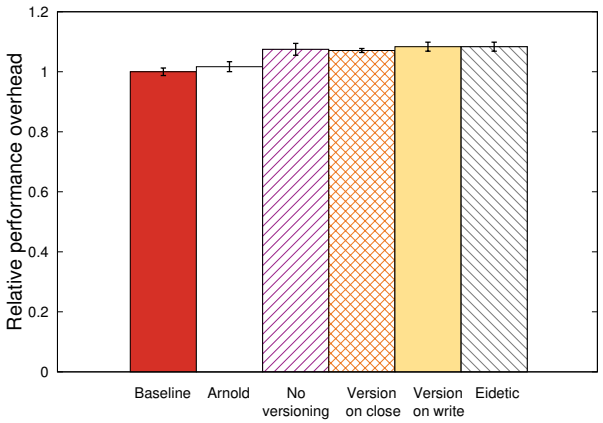


Figure 10: Performance overhead building libelf-0.8.9

shows the relative performance when we use Arnold to record the compilation with data stored in the baseline file system. This shows that the isolated cost of using deterministic record and replay is 2%. We then show relative performance when using Knockoff with its different versioning policies. The average performance overhead for using Knockoff ranges from 7% to 8%; the relative costs of different policies are equivalent within experimental error as shown by the overlapping 95% confidence intervals.

6.8 Log compression

Finally, we examine the benefit of using delta compression on logs of nondeterminism. Across all logs, delta compression reduces the bytes needed to store those logs by 42%. In comparison, chunk-based deduplication reduces the size of the logs by only 33%.

We find it interesting that chunk-based deduplication is more effective for compressing file data, whereas delta compression is more effective for compressing nondeterminism in the computation that produced that data. It is possible that restructuring the logs to make them more amenable to either delta compression or chunk-based deduplication could lead to further savings.

7 Conclusion

Operation shipping has long been recognized as a promising technique for reducing the cost of distributed storage. However, using operation shipping in practice has required onerous restrictions about application determinism or standardization of computing platforms, and these assumptions make operation shipping unsuitable for general-purpose file systems. Knockoff leverages recent advances in deterministic record and replay to lift those restrictions. It can represent, communicate, and store file data as logs of nondeterminism. This saves network communication and reduces storage utilization, leading to cost savings. In the future, we hope to extend the ideas in Knockoff to other uses; one promising target is reducing cross-data-center communication.

Acknowledgments

We thank the anonymous reviewers and our shepherd An-I Wang for their thoughtful comments. This work has been supported by the National Science Foundation under grants CNS-1513718 and CNS-1421441 and by a gift from NetApp. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, NetApp, or the University of Michigan.

References

[1] ADAMS, I. F., LONG, D. D. E., MILLER, E. L., PASUPATHY, S., AND STORER, M. W. Maximizing efficiency by trading storage for computation. In *Proceedings of the Workshop on Hot Topics in Cloud Computing* (June 2009).

[2] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 193–206.

[3] Apple time machine. `https://support.apple.com/en-us/HT201250`.

[4] AT&T Broadband monthly data allowance. `https://www.att.com/support/internet/usage.html`.

[5] Amazon Web Services (AWS) pricing. `https://aws.amazon.com/ec2/pricing/`.

[6] BENT, J., THAIN, D., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LIVNY, M. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation* (March 2004).

[7] Oracle Berkeley DB. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb`.

[8] BHATTACHERJEE, S., CHAVAN, A., HUANG, S., DESHPANDE, A., AND PARAMESWARAN, A. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *The Proceedings of the VLDB Endowment* (August 2015), 1346–1357.

[9] CHACON, S., AND STRAUB, B. *Pro Git (2nd Edition)*. Apress, November 2014.

[10] CHANG, T.-Y., VELAYUTHAM, A., AND SIVAKUMAR, R. Mimic: Raw activity shipping for file synchronization in mobile file systems. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services* (Boston, MA, June 2004), pp. 165–176.

[11] Comcast Broadband monthly data allowance. https://customer.xfinity.com/help-and-support/internet/data-usage-plan.

[12] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 285–298.

[13] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (Broomfield, CO, October 2014).

[14] Dropbox. http://www.dropbox.com.

[15] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M., AND CHEN, P. M. Execution replay on multiprocessor virtual machines. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2008), pp. 121–130.

[16] ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (September 2002), 375–408.

[17] Filesystem in Userspace. http://fuse.sourceforge.net/.

[18] Google drive. https://www.google.com/drive/.

[19] GORDON, M., HONG, D. K., CHEN, P. M., FLINN, J., MAHLKE, S., AND MAO, Z. M. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th International Conference on Mobile Systems, Applications and Services* (2015).

[20] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[21] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (2010).

[22] LEE, C., LEHOCZKY, J., RAJKUMAR, R., AND SIEWIOREK, D. On quality of service optimization with discrete QoS options. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium* (June 1999).

[23] LEE, Y.-W., LEUNG, K.-S., AND SATYANARAYANAN, M. Operation shipping for mobile file systems. *IEEE Transactions on Computers 51*, 12 (December 2002), 1410–1422.

[24] MASHTIZADEH, A. J., BITTAU, A., HUANG, Y. F., AND MAZIÈRES, D. Replication, history, and grafting in the ori file system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farminton, PA, October 2013), pp. 151–166.

[25] Microsoft onedrive. https://onedrive.live.com/about/en-us/.

[26] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS) 17*, 1 (March 1992), 94–162.

[27] MUMMERT, L., EBLING, M., AND SATYANARAYANAN, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995).

[28] MUNISWAMY-REDDY, K.-K., AND HOLLAND, D. A. Causality-based versioning. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (San Francisco, CA, February 2009).

[29] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (Boston, MA, May/June 2006), pp. 43–56.

[30] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (San Francisco, CA, March/April 2004), pp. 115–128.

[31] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Canada, October 2001), pp. 174–187.

[32] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 177–191.

[33] PARKER, D. S., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B. J., WALTON, E., CHOW, J. M., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistencies in distributed systems. *IEEE Transactions on Software Engineering SE-9*, 3 (May 1983), 240–247.

[34] QUINN, A., DEVECSERY, D., CHEN, P. M., AND FLINN, J. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation* (Savannah, GA, November 2016).

[35] rr: lightweight recording and deterministic debugging. http://www.rr-project.org.

[36] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. *SIGOPS Operating Systems Review 33*, 5 (1999), 110–123.

[37] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22*, 4 (December 1990), 299–319.

[38] SHILANE, P., HUANG, M., WALLACE, G., AND HSU, W. WAN optimized replication of backup datasets using stream-informed delta compression. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012).

[39] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), pp. 43–58.

[40] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Tech. Rep. TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.

[41] VAHDAT, A., AND ANDERSON, T. Transparent result caching. In *Proceedings of the 1998 USENIX Annual Technical Conference* (June 1998).

[42] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Long Beach, CA, March 2011).

[43] VRABLE, M., SAVAGE, S., AND VOELKER, G. M. Cumulus: Filesystem backup to the cloud. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (San Francisco, CA, February 2009), pp. 225–238.

[44] Wayback: User-level versioning file system for linux. `http://wayback.sourceforge.net/`.

[45] Xdelta. `http://xdelta.org/`.