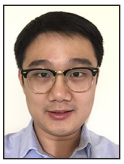
Knockoff
Cheap Versions in the Cloud

XIANZHENG DOU, PETER M. CHEN, AND JASON FLINN



Xianzheng Dou is a PhD student in computer science and engineering at the University of Michigan, Ann Arbor. In general, his research interests include file systems, operating systems, and distributed file systems. More specifically, he has been focusing on how to reduce communication and storage costs for distributed file systems and how to speed up computation via memorization. xdou@umich.edu

Peter M. Chen is an Arthur F. Thurnau Professor in the Computer Science Division at the University of Michigan. He is an ACM and IEEE Fellow and served as the Editor-in-Chief of *ACM Transactions on Computer Systems* from 2009–2013. In 2007, he received the ACM SIGOPS Mark Weiser Award "for creativity and innovation in operating systems research." His research interests include operating systems, computer security, and fault-tolerant computing. He is currently investigating how to improve software reliability for multicore computers and how to integrate new types of persistent memories into computer systems. He regularly teaches a senior course on operating systems and a first-year course on computer engineering. pmchen@umich.edu

Jason Flinn is a Professor of Computer Science and Engineering and Director of the Software Systems Laboratory at the University of Michigan, whose research interests include operating systems, distributed systems, and mobile computing. He is a fellow of the ACM, and his research has been recognized with an NSF CAREER award and eight Best Paper awards at SOSP, OSDI, ASPLOS, FAST, and MobiSys. jflinn@umich.edu

Cloud-based storage provides reliability and ease-of-management. Unfortunately, it can also incur significant costs for both storing and communicating data. These costs increase when systems retain past versions of files for data recovery, auditing, and forensic troubleshooting. While techniques such as chunk-based deduplication and delta compression have proven very effective in reducing bytes stored and sent over the network, further optimizations to these techniques are yielding increasingly incremental benefits. We argue that it is time to consider additional strategies for reducing storage costs. In our current work, we are demonstrating that one such strategy, deterministic recomputation of data, can substantially reduce the cost of cloud storage. Our distributed file system, Knockoff, selectively substitutes nondeterministic inputs for file data. Our results show that this reduces the cost of sending files to the cloud without versioning by 21–24%; the relative benefit is substantially greater when past versions are retained.

Deterministic Recomputation

Knockoff leverages an unconventional method for communicating and storing file data. In lieu of the actual data, it selectively represents a file as a log of the nondeterministic inputs needed to recompute the data (e.g., system call results, thread scheduling, and external data read by a process). With such a log, a cloud file server can deterministically replay the computation that originally produced the data to recreate the data. We call the observation that one can represent data generated by computation either by value or by the log of inputs needed to reproduce the computation the *principle of equivalence* (between values and computation); the principle has been observed and used in many settings such as fault tolerance and state machine replication.

Representing data as a log of nondeterminism leads to several benefits for 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.

Although similar ideas have been previously applied to distributed storage, the computation has either been assumed to be deterministic given its command line and file inputs [4] or given a specific sequence of user-interface events [1]. Unfortunately, neither a log of shell commands nor a log of user activity is sufficient to reproduce the computation of modern, general-purpose programs, especially due to the shift to multithreaded computation running on multiprocessors, as well as a growing diversity in execution environments and corresponding dependencies on operating systems, libraries, and installed application versions.

Knockoff uses deterministic record and replay to guarantee that data produced by all data-race free programs can be reproduced. Rather than capture a subset of nondeterministic inputs, it uses the Arnold [2] system to record all nondeterministic data entering each process that executes on a file system client, including the results of system calls (such as user and network input), the timing of signals, and real-time clock queries. Arnold enables deterministic replay of multithreaded programs by recording all synchronization operations (e.g., `pthread_lock` and atomic hardware instructions). This recording has minimal overhead (8% or less in our experiments). 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).

For example, 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 (rc=3) 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. Data read from the file system is not in the log since Knockoff is a versioning file system that 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 aggressive compression [2], 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. Since the log contains references to executable and shared library versions, as well as all interactions with the operating system, the complex environmental dependencies of an application are automatically resolved as part of the replay process. For instance, the replay starts from the same executable, loads the same versions of shared libraries, and sees the same results of IPC and network operations that were seen during recording.

	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

Additionally, just as deduplication and compression of file data can reduce bytes stored and sent over the network for file data, we have found that applying these techniques to logs of nondeterminism can also provide similar savings by exploiting similarities in computation across executions of the same application. In particular, Knockoff achieves an additional 42% reduction in bytes stored and communicated by using delta compression on the logs of nondeterminism.

Writing Data to the Cloud

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

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

$size_{log}$ is determined by compressing the log of nondeterminism for the application that wrote the file and measuring its size directly. To estimate $time_{replay}$, Knockoff records 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 [6]. 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 [5] 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 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 current version, and potentially retains the past version as described below.

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. If verification fails, it asks the client to ship the file data. Such races are rare since the replay system itself acts as an efficient data-race detector. All subsequent replays are guaranteed to produce the same data as the first replay, so once Knockoff verifies that the replay produces the desired data, it need not do so again.

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.

Storing Data in the Cloud

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 stores the current version of all files 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.

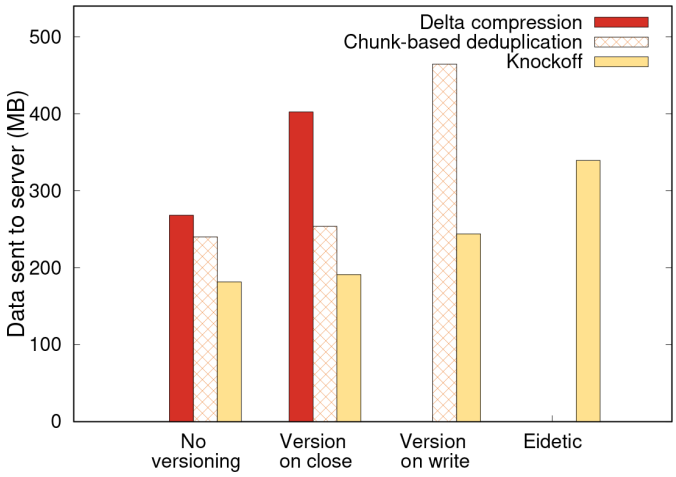


Figure 2: Total bytes sent to the server across all user study participants. We compare Knockoff with two baselines across all relevant versioning policies.

When replaying a log to regenerate data, Knockoff may find that some of the input files for the computation being replayed are also stored by operation rather than by value. In this case, it recursively replays those logs to reproduce the input data needed to regenerate the target data. Knockoff tracks such recursive dependencies in a data structure called the *version graph*. When storing data, it ensures that any path of recomputation in this graph does not exceed the materialization delay, and this guarantees that the total time to reproduce any file is no greater than that bound.

Fine-Grained Versioning

Past file versions have many uses: recovery of lost or overwritten data, reproduction of the process by which data was created, auditing, and forensic troubleshooting. These benefits increase as versions are retained more frequently. For instance, if versions are retained every time a file is closed, the user may have a snapshot of file data with each save operation. However, many applications only close files on termination, so versioning on every file write may be required to provide snapshots of intermediary states. However, such a policy would not capture intermediary states from modifications to memory-mapped files.

When storing and communicating file data by value, more frequent versioning substantially increases costs due to a greater amount of data sent over the network and saved to disk. However, when Knockoff represents file data by operation, its deterministic recomputation can produce *any* version of file data written by that computation at no additional cost. This means that Knockoff has much lower costs for retaining past versions of file data than traditional storage systems.

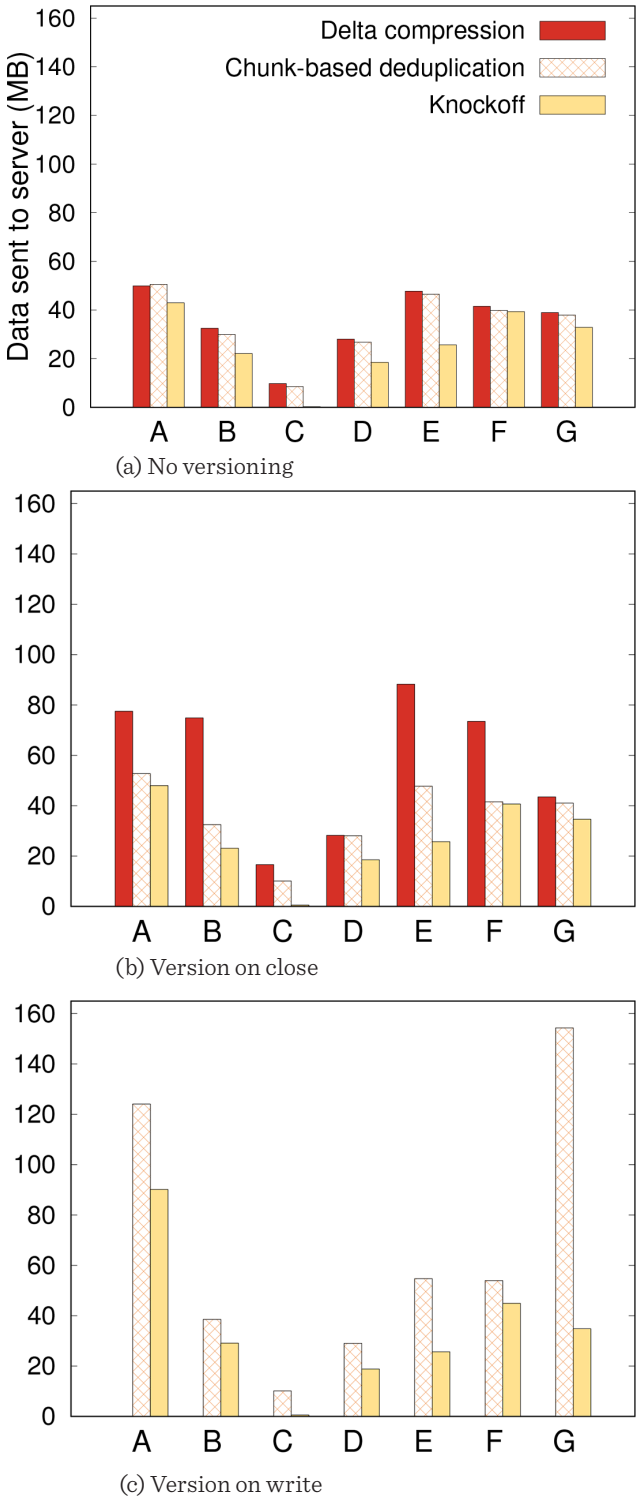


Figure 3: Bytes sent to the server for each individual user-study participant (A-G). We compare Knockoff with two baselines across all relevant versioning policies.

As a result, Knockoff currently supports four different versioning policies:

- **No versioning**: Knockoff retains only the current version of all files.
- **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.
- **Version on write**: Knockoff retains all past versions at write granularity.
- **Eidetic**: Knockoff retains all past versions at instruction granularity. It can reproduce versions of a memory-mapped file by replaying the computation up to a specified point and redoing the individual store instructions that modified the file.

User Study Results

As part of a detailed evaluation of Knockoff [3], we recruited eight graduate students to use Knockoff for software development tasks. We asked participants 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). Almost all files accessed during the study are stored in Knockoff (exceptions include the tmp directory and system configuration files), and almost all file modifications are therefore persisted in the cloud. One of the eight participants was unable to complete the programming assignment and quit right away. We show results for the seven participants who attempted the tasks; four of these finished successfully within the hour.

Figure 2 summarizes the results by aggregating the bytes sent to a cloud server by Knockoff and the baseline file systems across all seven users. Even without retaining past versions, 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 these baselines are already very effective in reducing bandwidth; without compression, this workload requires 1.9 GB of communication, so delta compression alone achieves an 86% reduction in network bandwidth, and chunk-based deduplication achieves an 87% reduction.

The benefit of Knockoff increases substantially as past versions are maintained more frequently. For instance, Knockoff reduces bytes sent by 47% compared to chunk-based deduplication for a `version on write` policy. In fact, versioning on write with Knockoff uses less bandwidth than the baselines without versioning.

At the limit, the eidetic policy, which can reproduce any past version even for memory-mapped files, is completely infeasible with current storage systems that store data by value. Knockoff can support this granularity of versioning while sending only 41% more bytes to the cloud than chunk-based deduplication *without versioning* in the user study and storing only 134% more bytes in the cloud to retain this state in another longitudinal study (not shown).

A surprising result from this study was that the effectiveness of Knockoff varied tremendously across users, as shown in Figure 3 (each individual study participant is labeled A-G in each graph). 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%. This shows the orthogonal nature of Knockoff's cost savings. When the mix of tools and workloads is better for operation shipping than it is for deduplication or compression, Knockoff produces large savings. In cases where operation shipping is not economical, Knockoff can detect this and revert to more traditional forms of bandwidth and storage reduction.

Summary

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-datacenter communication. We are also investigating whether it is feasible to generate logs of nondeterminism from which data can be reproduced by observing only a portion of those nondeterministic inputs and synthesizing likely values for the rest. This could represent a promising middle ground between Knockoff and prior operation shipping systems in which one could still guarantee that data can always be reproduced once a successful recomputation has been generated, but such guarantees could be achieved without running a full-scale deterministic recording system such as Arnold on each client.

Acknowledgments

This work has been supported by the National Science Foundation under grants CNS-1513718 and CNS-1421441 and by a gift from NetApp.

References

[1] T.-Y. Chang, A. Velayutham, and R. Sivakumar, "Mimic: Raw Activity Shipping for File Synchronization in Mobile File Systems," in *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services* (June 2004), pp. 165–176.

[2] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic Systems," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)* (October 2014): https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-devecsery.pdf.

[3] X. Dou, P. M. Chen, and J. Flinn, "Knockoff: Cheap Versions in the Cloud," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)* (February 2017): https://www.usenix.org/system/files/conference/fast17/fast17-dou.pdf.

[4] Y.-W. Lee, K.-S. Leung, and M. Satyanarayanan, "Operation Shipping for Mobile File Systems," in *IEEE Transactions on Computers,* vol. 51, no. 12 (December 2002), pp. 1410–1422.

[5] A. Muthitacharoen, B. Chen, and D. Mazières, "A Low-Bandwidth Network File System," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (October 2001), pp. 174–187: https://pdos.csail.mit.edu/papers/lbfs:sosp01/lbfs.pdf.

[6] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn, "JetStream: Cluster-Scale Parallelization of Information Flow Queries," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)* (November 2016): https://www.usenix.org/system/files/conference/osdi16/osdi16-quinn.pdf.