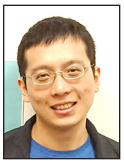
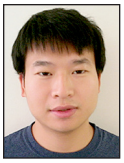
How to Not Copy Files

YANG ZHAN, ALEX CONWAY, NIRJHAR MUKHERJEE, IAN GROOMBRIDGE, MARTÍN FARACH-COLTON, ROB JOHNSON, YIZHENG JIAO, MICHAEL A. BENDER, WILLIAM JANNEN, DONALD E. PORTER, AND JUN YUAN



Yang Zhan recently completed his PhD at the University of North Carolina at Chapel Hill and now works as a senior engineer in the Operating Systems Kernel Lab at Huawei. yzhan@cs.unc.edu

Alex Conway recently completed his PhD at Rutgers University and is now a researcher at VMware. His interests focus on high-performance storage systems at the intersection of theory and practice. conway@ajhconway.com



Yizheng Jiao is a PhD student in the Computer Science Department at the University of North Carolina at Chapel Hill. He designs and implements efficient storage systems (e.g., in-kernel file systems and databases). yizheng@cs.unc.edu

Nirjhar Mukherjee is an undergrad at the University of North Carolina at Chapel Hill. nirjharm@gmail.com

Ian Groombridge is an undergrad at Pace University. igroombridge2010@gmail.com

Michael A. Bender is a professor of computer science at Stony Brook University. His research focuses on theory of algorithms and their use in storage systems. bender@cs.stonybrook.edu

Making logical copies, or clones, of files and directories is critical to many real-world applications and workflows, including backups, virtual machines, and containers. In this article, we explore the performance characteristics of an ideal cloning implementation; we show why copy-on-write induces a trade-off that prevents existing systems from achieving the ideal constellation of performance features; and we show how to achieve strong cloning performance in an experimental file system, BetrFS.

Many real-world workflows rely on efficiently copying files and directories. Backup and snapshot utilities need to make copies of the entire file system on a regular schedule. Virtual-machine servers create new virtual machine images by copying a pristine disk image. More recently, container infrastructures like Docker make heavy use of file and directory copying to package and deploy applications [5], and new container creation typically begins by making a copy of a reference directory tree.

Duplicating large objects is so prevalent that many file systems support *logical* copies of files or directory trees without making full *physical* copies. A physical copy is one where data blocks are duplicated, whereas a logical copy is one where data blocks may be shared. We call writable, logical copies ***clones***.

Writes to a logical copy should not modify the original file and vice versa. A classic way to maintain the content of a file is copy-on-write (CoW), where shared blocks are physically copied as soon as they are modified. Initially, this approach is also space efficient because blocks or files need not be rewritten until they are modified.

Many logical volume managers support CoW snapshots, and some file systems support CoW file or directory clones via `cp --reflink` or other implementation-specific interfaces. Many implementations have functional limitations, such as only cloning files, special directories marked as "subvolumes," or read-only clones. Nonetheless, we will refer to all these features as cloning.

Performance goal: nimble clones. An ideal clone implementation will have strong performance along several dimensions. In particular, clones should:

- be fast to create;
- have excellent read locality, so that logically related files can be read at near-disk bandwidth, even after modification;
- have fast writes, both to the original and the clone; and
- conserve space, in that the write amplification and disk footprint are as small as possible, even after updates to the original or to the clone.

We call a clone with this constellation of performance features ***nimble***.

Production clone implementations are not nimble. Nimble clones are the performance ideal, but CoW cloning does not yield nimble performance. This may seem surprising, especially given that CoW has been the de facto way to implement clones for decades.

Martín Farach-Colton is a professor of computer science at Rutgers University. His research focuses on theory of algorithms and their use in storage systems. martin@farach-colton.com

Bill Jannen is an assistant professor of computer science at Williams College. His research interests span a variety of topics, from computer science education to storage systems. jannen@cs.williams.edu

Rob Johnson is a senior staff researcher at VMware Research, where he works on the theoretical and applied aspects of high-performance storage systems. robj@vmware.com

Don Porter is an associate professor of computer science at the University of North Carolina at Chapel Hill. His research focuses on improving performance, security, and usability of computer systems. porter@cs.unc.edu

Jun Yuan is an assistant professor of computer science at Pace University. She is interested in building storage systems with solid theoretical foundation and with measured performance that matches the analysis. jyuan2@pace.edu

The Copy-on-Write Granularity Problem, or Why It's Hard to Achieve Nimble Clones

We begin by describing a simple implementation of CoW cloning in an inode-based file system. Although details will vary depending on the specifics of the file system, all existing production file systems share the CoW granularity trade-off illustrated in our simplified design below. This trade-off prevents these file systems from implementing nimble clones.

To clone a file from **a** to **b**, the file system can set up **b**'s inode to point to all the same data blocks as **a**'s inode, and both inodes are modified to mark all blocks as copy-on-write. With this approach, clones are cheap to create. In fact, if the file system uses extent trees to manage file blocks, it can mark entire subtrees of the extent tree as copy-on-write. This means that, to create the clone, the file system needs only to set up the old and new inodes to point to the same extent tree using copy-on-write.

This approach is also space efficient at first and preserves the locality of blocks within the file. If the blocks of the original file were laid out sequentially, then so are the clone's, so sequential reads from both will be fast. Note that this approach does not maintain inter-file locality: the blocks of clone **b** may be quite distant from the blocks of other files in **b**'s directory.

The challenge is to maintain space efficiency and good read locality as the files are edited.

Whenever the file system performs a write to a shared block of either file, the file system must allocate a new block and redirect the modified file's inode to point to the new data block.

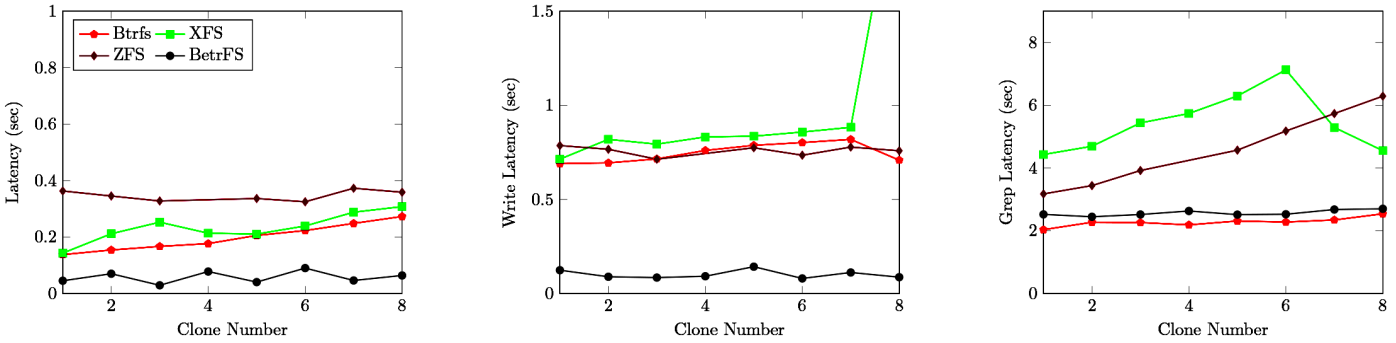
This simple but representative implementation of CoW exhibits a trade-off among space conservation, read locality, and write throughput. The main tuning parameter for CoW is the copy granularity. Copy granularity is the size of the data block that is copied when a file is modified. At one extreme, the entire file can be copied, and at the other, the system might only copy a sector on the device—typically 512 bytes or 4 KiB.

File-granularity CoW can have poor write throughput and space efficiency. If one makes a small change to a large file, this small write will incur the cost of copying the entire file and miss a significant opportunity to share a large portion of identical contents. File-granularity CoW favors read locality, but even this goal isn't quite met: if a small file is modified and copied, its placement in storage can cause inter-file fragmentation and, thus, low read throughput for some workloads.

At the other extreme, fine-granularity CoW, say at block granularity, will struggle to conserve locality. Over time, the blocks of a file can scatter across the storage device as they are allocated 4 KiB at a time. For example, consider a large file that is initially placed in a physically contiguous run of blocks, cloned, and then a series of small, random writes are issued to both files. As soon as one block in the middle of this run is modified, the block must be rewritten out-of-place. This block is now far from its neighbors in either the original, the clone, or both. Many file systems have heuristics for placing logically related blocks near each other at allocation time, but, in practice, this is not enough to prevent *aging* over the lifetime of the file system [2].

Put differently, small, random writes force simple CoW schemes either (1) to choose performance at write time and space efficiency (with fine-grained CoW) at the cost of read performance in the future, or (2) to choose read locality in the future (with coarse-grained CoW) at a higher write and space overhead.

BetrFS overcomes this trade-off by (1) aggregating small random application-level writes into large sequential disk writes and (2) using large CoW blocks. By aggregating small random writes, BetrFS ensures that random writes are fast and space efficient. By using large CoW blocks, it ensures that locality is maintained even as sharing is broken. See section "Nimble Clones in BetrFS" for details.



(a) Directory cloning latency (b) Small write latency (c) Grep latency

Figure 1: Latency to clone, write, and read as a function of the number of times a directory tree has been cloned. Lower is better for all measures.

Cloning Performance in Real File Systems

In this section, we use a microbenchmark to demonstrate the CoW-granularity trade-off in real file systems, and to show that BetrFS overcomes this trade-off. We then demonstrate how nimble clones can be used to accelerate real applications, such as container instantiation.

Dookubench: A Cloning Microbenchmark

To demonstrate the challenges to cloning performance in production file systems, we wrote a cloning microbenchmark, which we call Dookubench. Like its Star Wars namesake, it makes adversarial use of cloning. The benchmark begins by creating a directory hierarchy with eight directories, each containing eight 4-MiB files. Dookubench then proceeds in rounds. In each round, it creates a new clone of the original directory hierarchy and measures the clone operation's latency. It then writes 16 bytes to a 4 KiB-aligned offset in each newly cloned file—followed by a sync—in order to measure the impact of copy-on-write on writes. The benchmark then clears the file system caches and greps the newly copied directory to measure cloning's impact on read time. Finally, the benchmark records the change in space consumption for the whole file system at each step.

We use Dookubench to evaluate cloning performance in Btrfs, XFS, ZFS, and BetrFS. All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GiB RAM, and a 500 GB, 7200 RPM SATA disk, with a 4096-byte block size. The system runs 64-bit Ubuntu 14.04.5. Note that only BetrFS supports clones of arbitrary files and directories. We deal with the limitations of other file systems as follows. In Btrfs and XFS, we copy the directory structure in each round and use `cp --reflink` to create clones of all the files. For ZFS, we configure the root of the benchmark directory as a sub-volume, and use ZFS's volume snapshotting functionality to perform the clone.

FS	Δ KiB/round
Btrfs	176
XFS	32.6
ZFS	250
BetrFS	16.3

Table 1: Average change in space usage after each Dookubench round (a directory clone followed by small, 4 KiB-aligned modifications to each newly cloned file)

The write-granularity trade-off is illustrated clearly in Table 1 and Figure 1c. XFS uses relatively little space per round, suggesting it is using a small CoW block size. As would be expected of a CoW system with a small block size, Figure 1c shows that the amount of time required to scan through all the contents of the cloned directory degrades with each round of the experiment—after six clones, the grep time is nearly doubled. There appears to be some work that temporarily improves locality, but the degradation trend resumes after more iterations (not shown).

The Btrfs `grep` performance is much flatter, but this comes at the cost of much larger space usage per clone—Btrfs used an average of 176 KiB per clone, compared to 16.3 KiB for BetrFS and 32.6 KiB for XFS. Furthermore, its performance is not completely flat: Btrfs degrades by about 20% during the experiment. After 17 iterations (not presented for brevity), Btrfs read performance degrades by 50% with no indication of leveling off. ZFS is both space-inefficient, using 250 KiB per clone, and shows more than a 2× degradation in scan performance throughout the experiment.

Only BetrFS achieves low space per clone while maintaining locality, as shown by its flat performance on the `grep` benchmark. BetrFS uses 16 KiB per clone—half the space of the next-most-space-efficient file system (XFS)—and its read performance is competitive with the much less space-efficient Btrfs.

BetrFS excels at clone creation (Figure 1a) and small random writes to clones (Figure 1b). BetrFS's cloning time is around 60 ms, which is 33% faster than the closest data point from another

file system (the first clone on XFS) and an order of magnitude faster than the worst case for the competition. Furthermore, BetrFS 's clone performance is essentially flat throughout the experiment. ZFS also has flat volume-cloning performance, but not as flat as BetrFS. Both Btrfs and XFS file-level clone latencies, on the other hand, degrade as a function of the number of prior clones; after eight iterations, clone latency is roughly doubled.

In terms of write costs, the cost to write to a cloned file or volume is flat for all file systems, although BetrFS can ingest writes 8–10× faster. This derives from BetrFS's write-optimized design.

In total, these results indicate that BetrFS supports a seemingly paradoxical combination of performance features: clones are fast and space-efficient, and random writes are fast, yet preserve good locality for sequential reads. No other file system in our benchmarks demonstrated this combination of performance strengths, and some also showed significant performance declines with each additional clone.

Cloning Containers

Linux Containers (LXC) is one of several popular container infrastructures that has adopted a number of storage back ends in order to optimize container creation. The default back end (dir) does an `rsync` of the component directories into a single, chroot-style working directory. The ZFS and Btrfs back ends use subvolumes and clones to optimize this process. We wrote a BetrFS back end using directory cloning.

Table 2 shows the latency of cloning a default Ubuntu 14.04 container using each back end. Container instantiation using clones on BetrFS is 3–4× faster than the other cloning back ends, and up to two orders of magnitude faster than the rsync-based back ends. Interestingly, BetrFS is also the fastest file system using the rsync-based back end, beating the next fastest file system (Btrfs) by more than 40%.

Back End	File System	lxc-clone (s)
Dir	ext4	19.514
	Btrfs	14.822
	ZFS	16.194
	XFS	55.104
	NILFS2	26.622
	BetrFS	8.818
ZFS	ZFS	0.478
Btrfs	Btrfs	0.396
BetrFS	BetrFS	0.118

Table 2: Latency of cloning a container

Nimble Clones in BetrFS

This section overviews the four key techniques BetrFS uses to realize nimble clones. The interested reader can find a detailed explanation, as well as related work, in our recent FAST '20 paper [4].

BetrFS [1, 3] is an in-kernel, local file system built on a key-value store (KVstore) substrate. A BetrFS instance keeps two KVstores. The metadata KVstore maps full paths (relative to the mount-point, e.g., `/foo/bar/baz`) to `struct stat` structures, and the data KVstore maps {full path + block number} keys to 4 KiB blocks.

BetrFS is named for its KVstore data structure, the B$^\varepsilon$-tree [1]. A B$^\varepsilon$-tree is a write-optimized KVstore in the same family of data structures as LSM-trees (Log-Structured Merge-tree). Like B-tree variants, B$^\varepsilon$-trees store key-value pairs in leaves. A key feature of the B$^\varepsilon$-tree is that interior nodes buffer pending mutations to the leaf contents, encoded as *messages*. Messages are inserted into the root of the tree, and, when an interior node's buffer fills with messages, messages are *flushed* in large batches to one or more children's buffers. Eventually, messages reach the leaves and the updates are applied. As a consequence, random updates are inexpensive—the B$^\varepsilon$-tree effectively logs updates at each node. Note that these buffers are bounded in size to a few MiB, and buffers are never allowed to grow so large that they suffer from common pathologies in a fully log-structured file system. And since updates move down the tree in batches, the I/O savings grow with the batch size.

A key change needed to share data at rest is to convert the B$^\varepsilon$-tree into a B$^\varepsilon$-*Directed Acyclic Graph* (DAG). Nodes in the B$^\varepsilon$-DAG can be shared among multiple paths from the root to a leaf; sharing a sub-graph of the B$^\varepsilon$-DAG yields space-efficient clones. So far, this is a standard approach to copy-on-write. A nimble design is realized with four additional techniques.

Technique 1: Write Optimization. In order to avoid the granularity trade-off of CoW, we use buffers in a B$^\varepsilon$-DAG to accumulate small writes to a cloned file or directory. The key feature of write-optimization that contributes to nimble clones is "pinning" messages above a shared node in the B$^\varepsilon$-DAG. For example, if we clone a large file `foo` to `bar` and make a small modification to `bar`, that change is encoded in a message and written into the root of the tree, with destination `bar`. However, this message will not be flushed into a shared node in the B$^\varepsilon$-DAG, or else it would "leak" the change into the original file `foo`. Holding a small "delta" in the parent node is more space efficient than making a full copy for a small change, or even copying one leaf node. Instead, we wait until enough changes for `foo` or `bar` accumulate so that little of the remaining content is shared, and then we break that sharing by creating two unique, unshared copies of a node and repacking the contents, potentially recovering locality. We call this technique *Copy-on-Abundant-Write* (CAW).

Technique 2: Full-Path Indices. BetrFS maintains inter-file locality and supports arbitrary file and directory clones by using full-path indexing. BetrFS indexes all files and blocks by their full path, and paths are sorted in DFS (depth first search) traversal order. This means that all the paths for a sub-tree of the directory hierarchy are contiguous in the key space. As a result, a DFS traversal of the directory hierarchy will correspond to a linear scan of the key-space, which will translate into large sequential I/Os, since the BetrFS B$^\varepsilon$-tree uses 4 MiB nodes.

Furthermore, this means that cloning an entire sub-tree of the directory hierarchy corresponds to cloning a contiguous range of keys, all of which have a common prefix.

Technique 3: Lifting. As stated so far, key-value pairs encode full pathnames. So nodes or sub-graphs at rest will be shared but have incorrect, full-path keys along one of the shared paths. In our example above, nodes storing the key-value pairs backing `bar` will initially have `foo` keys. We observe that cloning a file or directory from **a** to **b** is essentially duplicating all the key-value pairs that start with **a** to new key-value pairs in which **a** has been replaced by **b** in each key.

Lifting removes a common prefix from the keys of a node and instead stores this prefix along with the pointer and pivot keys in the parent. For instance, if an entire B$^\varepsilon$-DAG leaf stores key-value pairs under directory `/home/user`, this common prefix would be removed from each key-value pair in that leaf, and instead the prefix is stored once in the parent, retaining only "short" pathnames in the child. With lifting, two parents with different directory prefixes can share a node, copy-on-write, and queries dynamically construct the full-path key based on the path taken through the graph to reach a given node.

Technique 4: Lazy Updates. In order to keep latency of a copy low, we must batch and amortize the cost of updates. First, we create GOTO messages that edit the B$^\varepsilon$-DAG itself as they are flushed. This is new; previously, all write-optimized dictionaries only batched changes to the *data*, not the data structure itself. Specifically, a GOTO message encodes a pointer that adds an edge to the graph, redirecting searches for a cloned key range to the source of the copy. These messages are flushed down the graph in a batch, and eventually become regular edges once they reach a target height.

The discussion to this point assumes that a cloned file or directory happens to be within a proper sub-graph in the B$^\varepsilon$-DAG; this may not be the case, as nodes in a B$^\varepsilon$-DAG do not have the same structure as the file system directory tree. Nodes in a B$^\varepsilon$-DAG pack as many keys (in key order) as needed to reach a target node size; thus a node may contain multiple small files packed into a 4 MiB node or a single 4 MiB chunk of a large file. Rather than immediately removing data outside of the cloned range, and making a proper sub-graph with the source prefix removed, we instead add additional bookkeeping to delay these edits.

Specifically, we augment lifted pointers with *translation prefixes*, which can specify both a prefix substitution for data at rest that has not already been handled by lifting and, implicitly, a range of keys in a child to ignore. In the example of cloning `foo`, the root of the sub-graph storing `foo` may also include keys for `fii` and `fuu`; a filter on the path for `bar` would specify that any query that follows this path should ignore keys without prefix `foo`. Similarly, if the sub-graph for `foo` has not yet lifted `foo` out of the children, a translation prefix along the path to `bar` would indicate that, when looking in the `foo` sub-graph, any keys that start with `foo` should be translated to have prefix bar.

Conclusion

This article demonstrates that a variety of file systems operations are instances of a *clone* operation, and the available implementations share the same copy-on-write-induced trade-off. This trade-off can be avoided by using write-optimization to decouple writes from copies, rendering a cloning implementation in BetrFS with the *nimble* performance properties: efficient clones, efficient reads, efficient writes, and space efficiency. The latency of the clone itself, as well as subsequent writes, are kept low by inserting a message into the tree. By making the changes in large batches, BetrFS conserves space. As data is copied on abundant writes, read locality is preserved and recovered by using full-path indexing to repack logically contiguous data into large, physically contiguous nodes. This unlocks improvements for real applications, such as a 3–4× improvement in LXC container cloning time compared to specialized back ends.

Acknowledgments

This research was supported in part by NSF grants CCF-1715777, CCF1724745, CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CCF1712716, CNS-1938709, and CNS-1938180. The work was also supported by VMware, by EMC, and by NetApp Faculty Fellowships.

References

[1] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan, "An Introduction to B^{ε}-trees and Write-Optimization," *:login;*, vol. 40, no. 5 (October 2015), pp. 22–28: https://www.usenix.org/system/files/login/articles/login_oct15_05_bender.pdf.

[2] A. Conway, A. Bakshi, Y. Jiao, Y. Zhan, M. A. Bender, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and M. Farach-Colton, "How to Fragment Your File System," *:login;*, vol. 4, no. 2 (Summer 2017), pp. 6–11: https://www.usenix.org/system/files/login/articles/login_summer17_02_conway.pdf.

[3] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. Porter, "BetrFS: Write-Optimization in a Kernel File System," *ACM Transactions on Storage*, vol. 11, no. 4 (November 2015), pp. 18:1–18:29: https://www.cs.unc.edu/~porter/pubs/a18-jannen.pdf.

[4] Y. Zhan, A. Conway, Y. Jiao, N. Mukherjee, I. Groombridge, M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, D. Porter, and J. Yuan, "How to Copy Files," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, pp. 75–89: https://www.usenix.org/conference/fast20/presentation/zhan.

[5] F. Zhao, K. Xu, and R. Shain, "Improving Copy-On-Write Performance in Container Storage Driver," 2016 Storage Developer Conference (SDC 2016): https://www.snia.org/sites/default/files/SDC/2016/presentations/capacity_optimization/FrankZaho_Improving_COW_Performance_ContainerStorage_Drivers-Final-2.pdf.