DenseFS: A Cache-Compact Filesystem

Zev Weiss
University of Wisconsin-Madison

Andrea C. Arpaci-Dusseau
University of Wisconsin-Madison

Remzi H. Arpaci-Dusseau
University of Wisconsin-Madison

Abstract

As nonvolatile memory technologies with access latencies comparable to DRAM proliferate, the CPU performance of previously storage-bound workloads becomes increasingly important. In this paper we examine the effects of the filesystem on cache behavior, a key aspect of CPU performance. We then develop DenseFS, a specialized filesystem that aims for a highly compact cache footprint and hence tries to minimize its cache pollution and the performance penalties it incurs. We find that DenseFS is effective in reducing the performance penalty of filesystem operations on user code, and can achieve dramatic reductions in cache miss rates as compared to existing filesystems.

1 Introduction

Storage device speeds have increased considerably with the increasing adoption of flash in applications that previously had employed hard disk drives [9, 12]. With the increasing availability of non-volatile memory (NVM) technologies [6, 8, 14], systems with persistent storage accessible with DRAM-like latencies may soon be widespread. With these dramatic improvements in the performance of storage hardware, the overhead incurred by the software managing it becomes more and more significant and storage-intensive applications that were previously I/O-bound become increasingly CPU-bound. This transition has led to research efforts into techniques like kernel-bypass filesystems [2, 13, 16–18] and in-device filesystems [10].

One of the most important factors in the CPU performance of a workload is its hit rate in the CPU cache [1, 5, 11], a hardware resource shared by both the application and the operating system's storage stack. This sharing means that in addition to the performance of filesystem code itself, the design and implementation of performance-conscious filesystems should also give consideration to the effects of cache pollution – that performing filesystem operations perturbs the delicate cache state needed to achieve good performance in executing non-filesystem code.

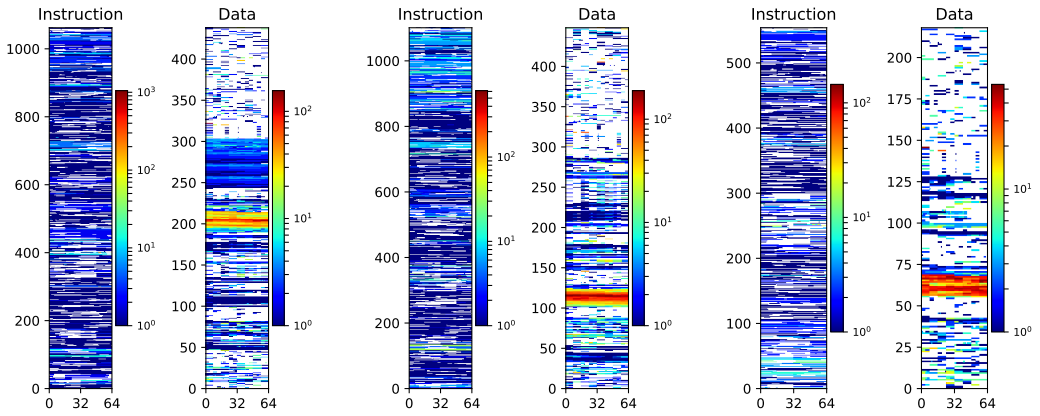
However, filesystem research thus far has spent little effort on this facet of the storage stack. Software design decisions both small and large, as well as phenomena such as code alignment that are not typically consciously decided by software developers (but can be controlled by a programmer who is aware of them), can play a significant role in a filesystem's cache behavior.

In order to examine and experiment with its impact on application performance, in this paper we study the cache footprints and access patterns of different Linux filesystems. We then develop an experimental filesystem, DenseFS, with the explicit aim of having a very compact cache footprint, and evaluate the performance benefits of the reduced pollution of application cache state that this smaller footprint provides. With targeted microbenchmarking we find that in comparison to an array of existing Linux filesystems, DenseFS can dramatically reduce the performance impact of the cache pollution caused by filesystem operations, in some cases reducing a 150% overhead to merely 20%. Using a real-world program, we find that using DenseFS in place of other existing filesystems can achieve a 37-65× reduction in L1 instruction cache misses, providing a 13% to 18% improvement in user-mode CPU performance.

The remainder of this paper is organized as follows. In Section 2 we investigate the cache behavior of existing Linux filesystems; in Section 3 we present the design and implementation of DenseFS; in Section 4 we evaluate the performance of DenseFS in comparison to other filesystems; finally, Section 5 concludes.

2 Filesystem Cache Access Patterns

We begin with an investigation of cache behavior in existing Linux filesystems. By scripting `gdb` attached



(a) `creat` (b) `unlink` of a 4KiB file (c) `rename`

Figure 1: Cachemaps of three operations on xfs.

to the kernel running in a virtual machine, we collect instruction-level dynamic traces of btrfs, ext4, f2fs, tmpfs, and xfs performing a variety of metadata operations. We trace each system call from the first kernel instruction to the last one before it resumes user-mode execution. For each instruction, we record: (1) the address and size of the instruction; (2) the addresses and sizes of any data memory accesses performed by the instruction; and (3) the full symbolic stack backtrace (the function name, source file, and line number for each stack frame).

Our first analysis processes these traces by aggregating all instruction and data memory accesses at byte granularity and counting the number of times each individual byte is accessed. We continue along the path of prior research in using heatmaps for visualizing cache access patterns [3, 15, 19] with a special heatmap we term a *cachemap* (see Figure 1). Each row of cells in a cachemap represents a single cache line (64 bytes), with each cell representing one byte of memory. The vertical axis serves simply to order cache lines by virtual address, though it is not generally contiguous (only cache lines that were accessed at least once are shown). The color of each cell provides a log-scale indication of how many times that byte was accessed[1] throughout the entire trace (with white representing the special value zero).

These cachemaps provide us with a starting point from which we can observe some general trends. First, the instruction cache footprint is typically about twice the size of the data cache footprint. Relative to the size of the first-level caches in current x86 processors (32KiB, or 512 64-byte lines), both are large enough to significantly perturb, if not displace entirely, warm userspace L1 cache state built up by an application.

[1]The program that generates these cachemaps also offers an interactive mode in which a user can click on a cell to see the full backtrace of every point at which that byte was accessed, making it easier to identifiy opportunities for potential optimizations.

Secondly, many data cache accesses are relatively wasteful in that they drag an entire line into the cache (displacing another one) only to provide a small handful of bytes, often to a single memory access. Accesses of this sort exhibit neither the spatial nor the temporal locality for which caches are optimized, and hence make very poor use of them.

Thirdly, instruction accesses, due to execution being inherently sequential by default, are somewhat less wasteful of cache resources in that a smaller number of bytes in each cache line go unused on average. However, despite this spatial locality, the prevalence of dark blue cells in the instruction cachemaps indicate that there is relatively little temporal locality (reuse of already-cached instructions); given the larger size of the instruction cache footprint this is still not a particularly effective use of hardware resources.

Due to its larger size, we focus first on optimizing instruction cache footprint. The low-level nature of instruction traces, however, makes it difficult to discern the major sources of that footprint. In order to gain a better understanding of this, we condense our instruction traces into *coarse-grained stack traces* or *cgstacks*, simplified views of the stack backtrace of a given instruction, and visualize them in the form of a flame graph [7].

Given an instruction's stack backtrace, we transform it into a cgstack by mapping each frame, progressing from callers to callees, to one of a set of designated code categories based on the file in which that function is defined (for example, functions in `mm/slab.c` are mapped to the "malloc" category, while `fs/file.c` is mapped to the "vfs" category). If the category classification of a given stack frame has not yet been seen in the corresponding cgstack thus far, that category is then added to the top of the cgstack. The result is effectively a high-level statement about the provenance of each instruction. For ex-

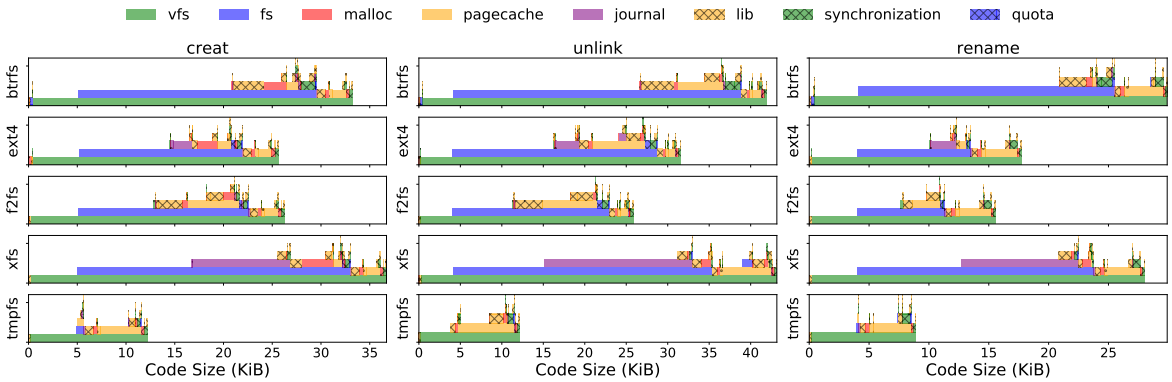


Figure 2: Cgstack flame graphs of code footprints.

ample, for a given instruction that statement may be that the instruction's presence in the trace is attributable to page cache code called by VFS code.

After transforming each instruction's stack trace into a cgstack in this way, we then weight each cgstack by the size of the instruction and aggregate the data together, producing a flame graph for each trace (see Figure 2).

These cgstack flame graphs show some variation between different filesystems, but make it clear that common, non-filesystem-specific infrastructure such as the VFS and page cache play a large role in overall code footprint. Armed with this knowledge, we set out to construct a new filesystem with the aim of maximizing cache density, in part by keeping it disentangled from the conventional filesystem framework. The resulting filesystem is called DenseFS, and is detailed in Section 3.

3 DenseFS

DenseFS is a small in-memory Linux filesystem implemented in approximately 2500 lines of code. Given the results of our analysis in Section 2 showing that the VFS and page cache code are significant contributors to the large code footprints of existing filesystems, DenseFS is not integrated into the "normal" Linux VFS layer and does not use its page cache. This is at the root of its primary practical difficulty: the standard file-access system calls (`open`, `read`, `unlink`, etc.) cannot be used to access it. Instead, it offers its own parallel set of system calls (`dfs_open`, `dfs_read`, `dfs_rename`, and so forth) with the same arguments, but which operate on files in the DenseFS namespace. DenseFS file descriptors are distinct from (and not interchangeable with) normal file descriptors, but otherwise operate similarly. Alongside its existing file descriptor table and working directory, each process thus gains a separate DenseFS file descriptor table and DenseFS working directory.

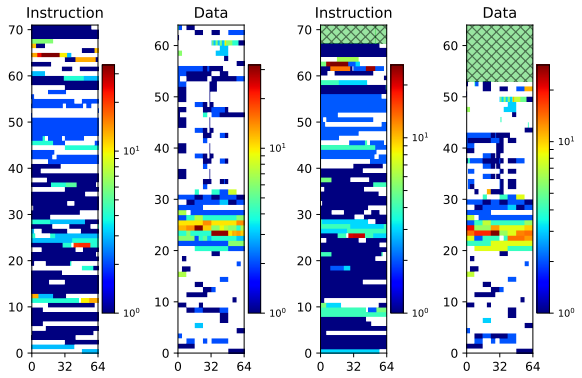
Within its set of special system calls, however, DenseFS has familiar features. Directory entries, inodes, and a superblock are represented with C `structs`, with pointers linking them together in the same overall structure found in most Unix-style filesystems. These structs are allocated in memory, but instead of using the general-purpose in-kernel memory allocation routines (Linux's `kmalloc` family of calls), it instead performs one large allocation for the entire (fixed) capacity of the filesystem when it is mounted and then allocates its own internal structures within that region of memory (mimicking what would be done in a true NVM filesystem).

3.1 Data Cache Compaction

In keeping with DenseFS's aims of being compact, some familiar structures are implemented differently than in conventional filesystems, in particular its inode. A straightforward inode structure for an in-memory filesystem like DenseFS might closely resemble the `stat` struct used in the standard `stat` system call, and indeed this was our initial starting point with DenseFS. With a few additional fields needed internally (a spinlock, a reference count for open files, and a union of pointers for directory entries and file data), this simple implementation, however, yields a 112-byte inode – larger than desired for a cache-dense filesystem.

With that starting point we made a few simple changes to save space spent on timestamps: we replaced the bulky 16-byte `struct timespec` with the Linux kernel's internal 8-byte `ktime_t`, and removed the `atime` member entirely, since access times are rarely actually used by applications (filesystems are frequently mounted with the `noatime` option anyway). This saved 32 bytes by reducing the space spent on timestamps from 48 bytes to 16.

Inode numbers are also relatively little-used, though unlike atime the only information they encode is a unique identifier, and thus can be removed without any compromise of functionality or semantics. Instead of storing an inode number in each inode, DenseFS's `stat` call



(a) base (b) packed

Figure 3: Cachemaps of the `creat` operation on DenseFS, before and after manual cache-compaction optimization. The hatched green regions near the top of (b) indicate cache footprint eliminated by the optimizations described in Section 3.

instead populates the `st_ino` field with a value derived from the in-memory address of the inode itself. In order to allow for these synthetic inode numbers to remain persistent (were DenseFS operating on real non-volatile memory), we subtract the base address of the DenseFS memory region to form an offset instead of a raw pointer value, and then XOR this offset with a secret key stored in the DenseFS superblock in order to avoid leaking potentially-sensitive metadata to userspace [4]. This saves eight bytes in the DenseFS inode struct.

The next inode compaction change we applied to DenseFS is based on the observation that the user, group, and mode fields contain very little entropy – even in filesystems containing many millions of files, there may be only a few hundred unique combinations of these three fields, so encoding this near-duplicate information in every individual inode is a very inefficient use of space. In DenseFS we thus compress this information by keeping a filesystem-wide table of `<uid, gid, mode>` tuples and replacing the corresponding three entries in the inode struct with a single 16-bit index into this table. This saves another 10 bytes, and along with some padding bytes saved by the the previous changes and reordering a few fields, achieves an important goal: at 56 bytes, the DenseFS inode struct is now small enough to be contained entirely in a single cache line. An example of the resulting decrease in cache footprint can be seen in the cachemaps in Figure 3.

3.2 Instruction Cache Compaction

To compact DenseFS's code footprint, we first traced its execution of various calls and produced corresponding cachemaps as in Section 2. Guided by these cachemaps, we then applied three varieties of manual adjustments to help the executed code fit into fewer cache lines; Figure 3 shows the resulting cachemaps.

Function alignment: This is the most frequently applicable and hence the most impactful technique. An excellent example of it is found in the function `current_kernel_time64`, used in updating inode timestamps. The function's code is only 58 bytes long, short enough to fit in a single cache line, but its starting address is offset from the cache-line boundary such that it spills over into the next line, causing its execution to displace one more line than it truly requires. By annotating it to be aligned on a 64-byte boundary, we avoid this pitfall and keep it contained in a single cache line. It would be simple to use a compiler flag to apply this alignment constraint globally to all functions, but this is not necessarily always beneficial, as will be shown in our discussion of function ordering below.

Branch hinting: The opportunity for this optimization arises when the compiler arranges code suboptimally for a conditional such as an `if` block. Consider a simple example with an `if` block with a small body and no `else` clause. A straightforward compilation of the code might put the body of the `if` block "inline" with the surrounding code preceded by a conditional branch that skips over it when the condition is false. If the condition is rarely true, however, this results in wasted space in the instruction cache – the bytes for those instructions are brought into the cache alongside their neighboring instructions, but are never executed. If the bias of the condition is known, a more optimal compilation would instead place the body of the `if` block in a relatively far-off location after the main "hot" body of the function and branch to it (and then back) in the unlikely case that its condition is true. By identifying occurrences like this (which are visible as small gaps of white in our cachemaps), we can sometimes add appropriate annotations to such `if` conditions and squeeze out a few more precious bytes of wasted cache space.

Function ordering: In one case we observed a cluster of three functions, one 30 bytes, one 37, and one 28 bytes (`strcpy`, `strcmp`, and `strlen`, respectively). Despite totaling only 95 bytes, they nevertheless spanned four cache lines – 256 bytes worth of space. One of the two "wasted" lines was due to suboptimal alignment of `strlen` causing its code to spill onto a second line, but even after addressing that the trio of string functions that should have fit easily in two lines still consumed three. Despite being in the same source file, their relatively distant locations within that file led to the corresponding layout in memory not condensing them together as would be desirable for compactness. In this case, cache-line-aligning all three functions individually would still

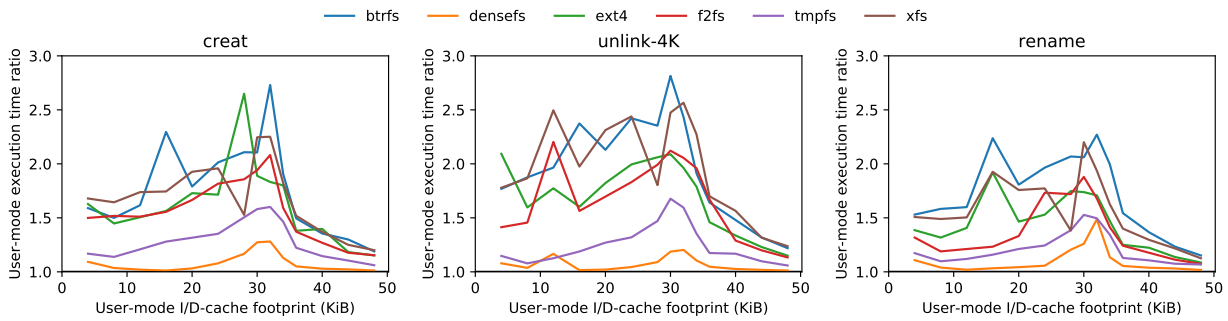


Figure 4: Microbenchmark performance results. The vertical axis shows the relative increase in time spent executing user-mode code when regular calls to the given system call on the given filesystem are inserted (i.e. the performance penalty of the syscall on user-mode execution). The horizontal axis shows the data and instruction cache footprints of the user-mode code executed between system calls.

reduce cache density by the same token – separating closely-related pieces of code. By simply reordering the functions to bring them together in the source file that defines them (`lib/string.c`), we were able to achieve the desired result of fitting all three into two cache lines.

4 Evaluation

We evaluated DenseFS's effectiveness in reducing overall cache pollution using a finely-parameterized synthetic microbenchmark to measure system call impact on user-mode CPU performance. We have also performed some preliminary experiments running a real application; both are presented in this section. All measurements were taken with an Intel Xeon E5-2670 CPU running a 4.13-series Linux kernel.

Microbenchmark results: Our microbenchmark tool exercises a single system call at a time, and offers the ability to execute an amount of user-mode "think-time" code in between each instance of the system call. This user-mode code is JIT-compiled before the main loop, and is parameterized to allow adjustment of its instruction and data cache footprints. The microbenchmark reports fine-grained performance statistics for the system call and the user-mode code independently.

Using this tool, we executed system calls and measured the performance of the user code while varying its cache footprint, and compare the results against the performance of executing the same user code with no system calls at all. This allows us to directly measure the performance impact on user-mode execution of the system call. Figure 4 shows the results, with DenseFS consistently incurring the smallest penalty on user-mode performance.

Preliminary application results: To evaluate DenseFS's performance on a real-world program, we ran version 3.1 of GNU `grep` over a 750MB directory tree containing 242,272 files, using an `LD_PRELOAD` library to redirect its system calls to their DenseFS equivalents. Using `perf stat`, we found that DenseFS is highly effective at reducing L1 instruction cache misses. Whereas xfs suffered 84.1M misses on this workload (the most of the five other filesystems tested) and tmpfs 49.0M (the least), DenseFS incurred only 1.3M, a reduction of 97% relative to tmpfs. This improvement allowed `grep`'s user-mode IPC to increase 13% over tmpfs and 18% over xfs. These results are promising, though further evaluation on real applications will be necessary.

5 Conclusion

We have shown with DenseFS that it is possible to implement a filesystem with a much smaller cache footprint than those of existing filesystems. Further, we have seen that the resulting reduction in cache pollution has a significant positive effect on the performance of user-mode application code. The implementation of DenseFS has made some trade-offs in functionality in order to achieve this small cache footprint; an interesting challenge to consider in further research on this topic would be how to eliminate some of these compromises (or reduce their negative effects) while maintaining as much as possible the compactness that DenseFS strives for.

Acknowledgements

We thank the anonymous reviewers and the members of ADSL for their valuable input. This material was supported by funding from NSF grants CNS-1421033 and DOE grant DESC0014935. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1999), VLDB '99, Morgan Kaufmann Publishers Inc., pp. 266–277.

[2] CAULFIELD, A. M., MOLLOV, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing Safe, User Space Access to Fast, Solid State Disks. In *ASPLOS XVII: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ACM. 415125.

[3] CHOUDHURY, A. I. *Visualizing Program Memory Behavior Using Memory Reference Traces*. PhD thesis, University of Utah, 2012.

[4] COOK, K. Kernel Address Space Layout Randomization. Linux Security Summit, 2013.

[5] FERDMAN, M., ADILEH, A., KOCBERBER, O., VOLOS, S., ALISAFAEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., AND FALSAFI, B. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 37–48.

[6] FOONG, A., AND HADY, F. Storage as fast as rest of the System. In *2016 IEEE 8th International Memory Workshop* (Paris, France, May 2016).

[7] GREGG, B. The Flame Graph. *Queue 14*, 2 (Mar. 2016), 10:91–10:110.

[8] HADY, F. T., FOONG, A., VEAL, B., AND WILLIAMS, D. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE 105*, 9 (2017), 1822–1833.

[9] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2010), FAST'10, USENIX Association, pp. 7–7.

[10] KANNAN, S., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., WANG, Y., XU, J., AND PALANI, G. Designing a True Direct-access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2018), FAST'18, USENIX Association, pp. 241–255.

[11] LEBECK, A. R., AND WOOD, D. A. Cache Profiling and the SPEC Benchmarks: A Case Study. *Computer 27*, 10 (Oct. 1994), 15–26.

[12] LEE, C., SIM, D., HWANG, J.-Y., AND CHO, S. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2015), FAST'15, USENIX Association, pp. 273–286.

[13] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)* (Broomfield, Colorado, Oct. 2014).

[14] STRUKOV, D. B., SNIDER, G. S., STEWART, D. R., AND WILLIAMS, R. S. The missing memristor found. *Nature 453* (2008), 80–83.

[15] VAN DER DEIJL, E., KANBIER, G., TEMAM, O., AND GRANSTON, E. D. A Cache Visualization Tool. *Computer 30*, 7 (Jul 1997), 71–78.

[16] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 14:1–14:14.

[17] VOLOS, H., AND SWIFT, M. Storage Systems for Storage-Class Memory. In *Proc. of Annual Non-Volatile Memories Workshop (NVMW'11)* (2011).

[18] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)* (Newport Beach, California, Mar. 2011).

[19] YU, Y., BEYLS, K., AND D'HOLLANDER, E. H. Visualizing the Impact of the Cache on Program Execution. In *Proceedings Fifth International Conference on Information Visualisation* (2001), pp. 336–341.