HetFS: A Heterogeneous File System for Everyone

Georgios Koloventzos[1], Ramon Nou*[1], Alberto Miranda*[1], and Toni Cortes[1,2]

[1] Barcelona Supercomputing Center (BSC)
Barcelona, Spain
[2] Universitat Politècnica de Catalunya
Barcelona, Spain
georgios.koloventzos,ramon.nou,alberto.miranda,toni.cortes}@bsc.es

Abstract

Storage devices have been getting more and more diverse during the last decade. The advent of SSDs made it painfully clear that rotating devices, such as HDDs or magnetic tapes, were lacking in regards to response time. However, SSDs currently have a limited number of write cycles and a significantly larger price per capacity, which has prevented rotational technologies from begin abandoned. Additionally, Non-Volatile Memories (NVMs) have been lately gaining traction, offering devices that typically outperform NAND-based SSDs but exhibit a full new set of idiosyncrasies.

Therefore, in order to appropriately support this diversity, intelligent mechanisms will be needed in the near-future to balance the benefits and drawbacks of each storage technology available to a system. In this paper, we present a first step towards such a mechanism called HetFS, an extension to the ZFS file system that is capable of choosing the storage device a file should be kept in according to preprogrammed filters. We introduce the prototype and show some preliminary results of the effects obtained when placing specific files into different devices.

1 Introduction

Storage devices have shown a significant evolution in the latest decade. As the improvements in the latencies of traditional hard disk drives (HDDs) have diminished due to the mechanical limitations inherent to their design, other technologies have been emerging to try and take their place. For instance, NAND-based solid state drive (SSD) technology has been extremely successful in improving I/O latency and bandwidth, and this has led to SSD devices often being incorporated into the storage stack as a caching tier for HDD-based storage systems, and also to being used as the principal data repositories. This, in turn, has forced any major applications that were bound by access times (such as databases [3]), to change in order to adapt to this new technology. Nevertheless, completely replacing HDDs by more efficient SSDs can be economically prohibitive due the

* *The second and third authors contributed equally to this work.*

Koloventzos, G., Nou, R., Miranda, A., Cortés, A. HetFS: A heterogeneous file system for everyone. A: International Conference on High Performance. "High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS: Frankfurt, Germany, June 18-22, 2017: revised selected papers". Frankfur: Springer, 2017, p. 691-700.
The final publication is available at https://link.springer.com/chapter/10.1007/978-3-319-67630-2_49

larger cost per capacity of the latter. More importantly, however, NAND-based SSDs have a limited number of write cycles and, in fact, recent researches on long-term SSD usage in data warehouses have proved that, after intensive usage, the SSDs degrade so much that response times may equal those of HDDs [5, 6].

In addition to SSDs, Non-Volatile Memory (NVM) technology is currently being researched as a better alternative. The different NVM technologies being explored typically exhibit faster I/O latencies than SSDs, which are closer to those of DRAM rather than to NAND-based devices. As such, current research efforts are focusing on whether these devices should be used as an extension of DRAM or included as an additional (persistent) caching layer to the storage stack [14].

Moreover, despite the recent advancements in SSDs and NVMs, the technological development of HDDs has not stopped. For instance, hard drives featuring an Helium-filled enclosure were recently introduced to the market since the gas density allows for more platters and a higher rotational speed of up to 19,000rpms [22]. Shingled Magnetic Recording (SMR) [1] is also starting to find its way to customers, since it allows for a higher track density and increased capacity at similar cost.

Thus, in the near future, file systems will need to cope with a myriad of storage devices, each with particular performance and capacity characteristics, and each suitable to certain types of I/O workload. Current file systems, however, typically distribute data into available devices by placing them into a hierarchy according to performance, and using prefetching and multi-tier caching algorithms to reduce I/O latency.

This, however, typically disregards other considerations such as extending the life of devices such as SSDs through wear leveling, or tailoring a file's data distribution according to the usage that applications make of it. For instance, software engineers typically rely on writing to a file as barrier or as an atomic operation, a usage which is crucial for the resilience and for synchronization of applications [4]. This access pattern can represent a significant disadvantage for SSDs since it will wear the medium faster. Similarly, the OS libraries are primarily read dominant [19, 2] and could be classified according to how often they are accessed, placing the rarely-used into an HDD for cold storage and the more commonly used ones into an SSD for improved performance. Multimedia files i.e., RIFF format, can be split. The first part with all the information of the file in a fast medium and the rest that accessed mostly sequentially to a rotating one. Lastly, intra-file formats could also be exploited by placing each file section into the storage device more suitable for the expected access patterns. Therefore, in order to support the diversity in storage media, file systems will need to provide intelligent algorithms that (1) appropriately quantify and model the benefits and drawbacks of each available storage device; (2) capture the more typical patterns that applications use to access data; and (3) use this information to create a tailored dynamic data distribution that optimizes the usage of the available hardware.

In this paper, we present *HetFS*, an extension to the ZFS file system that includes a component to capture information about file usage, and a simple decision making mechanism that uses this information to decide, according to a user-provided classification, the storage device where a file should be placed. We introduce the *HetFS* prototype and show some preliminary results measured by applying different precomputed file distributions to the kernel's boot process. The results offer an insight to the expected ZFS overhead added by the new mechanisms and showcase the potential benefits of such distributions. Please note that these modifications are meant as a first step towards a more complex feedback loop where *HetFS* will automatically capture file usage information and will use this information (allowing some degree of tuning from the user) to produce a data distribution that is optimized w.r.t. a file's more common access patterns and the features of the available storage devices.

The remainder of the paper is organized as follows: Section 2 describes the modifications made to ZFS in order to capture usage information and implement the user-provided file distributions. Section 3 describes our experiments regarding boot times with different file distributions. Related work is discussed in Section 4, and Section 5 concludes with our findings.

2 Heterogeneous File System

This section discusses the modifications done to ZFS in order to support the *HetFS* file forwarding mechanism. We chose to implement *HetFS* as an extension of ZFS[3] because this file system offers facilities to manage both the physical and the logical layers [16]. While historically file systems have been constructed on top of a single physical device, ZFS manages physical storage by means of storage pools (or *zpools*), which can be created using multiple and heterogeneous devices such as HDDs, SSDs, NVM or even tapes. A zpool describes the physical characteristics of the storage devices that compose it (called *vdevs*), and acts as an arbitrary data store from which file systems can be created. By leveraging this feature as an extension of ZFS, *HetFS* is able to produce a file classification based on access patterns and later use this information to guide requests to a specific storage device within a zpool.

2.1 File Classification

File classification is done by modifying the ZFS Posix Layer (ZPL), which is the ZFS layer responsible for interfacing between the VFS and the underlying ZFS data management layers. This layer still has enough semantic information about which file is being accessed by a `read()` or `write()` operation, and also offers enough detail to allow us to track the access to individual data blocks. Thus, we include a red-black tree in the ZPL where each node contains two separate linked lists for read and write requests.

[3] Given that ZFS is proprietary software, we used a fork of OpenZFS [11] named ZFS on Linux (ZoL) [24]. For clarity we will keep referring to it as ZFS.

The information traced is inserted into the red-black tree by a specialized kernel thread after a request has returned without errors, in order to not interfere much with it. Currently, the data consists of the file name, the offset of the request, the length, and the type, that are extracted from the request. We also capture the time when the request arrived, and use this information to merge small requests into bigger ones if the current captured time is close to the previously stored one, and also if the offset is contiguous to the previous one. This approach gives us an insight on how files are accessed from applications, and also allows us to track if particular part of a file is accessed more which will help us to assess the access patterns for each individual file in future research.

Currently, the analysis of the collected information is done post-mortem, and the final decision of the available vdevs should store a particular file is left to the user. This decisions can be communicated to *HetFS* by means of a custom *procfs* interface with some pre-configured characteristics. In the future the operating system will conduct an automated analysis of these information to make an informed decision on which storage medium a file should be placed. If a manual decision has been made the analysis will not be taken into consideration.

2.2 Device Selection

Files can be typically classified by their access patterns: for instance, multimedia files are most likely to be accessed sequentially, and documents created from word processors follow complex internal structures which makes parts of the file more likely to be accessed with different frequencies and patterns than others. This means that the former would benefit from a ZFS vdev optimized for sequential access, whereas the latter would benefit from a ZFS vdev optimized for random accesses. Other files, like bitmaps, indexes, and even the file system's metadata would be better stored in a ZFS vdev that could benefit from byte addressability (e.g. NVM).

In order to forward I/O requests to the desired vdev, we modify the ZFS Block Allocation mechanism to use the analyzed information produced by the file classification mechanism, which is conveyed to *HetFS* through the aforementioned *procfs* interface. Information about the chosen vdev for a file is encoded into the ZFS equivalent of VFS' i-nodes, so that it can be propagated to all the necessary ZFS layers, and is then used to allocate *ZFS block pointers* into the appropriate vdev. Since the standard ZFS Block Allocation strategy relies on dynamic striping to maximize bandwidth, we modify the vdev selection algorithm to simply choose the device encoded into the file. In the future, however, this selection will also consider other factors like the vdevs performance, their optimal access mode as well as any limiting features. We also leverage existing code [23] by the ZFS team to place metadata into SSDs and extend it to several vdevs.

Note that, currently, a user or system administrator could decide to move files that need a lower access latency to a SSD for faster I/O bandwidth. If these files were write-intensive, it would decrease the durability of the SSD but the file would actually be served faster. These kinds of compromises would need to

be decided either by the administrator or automatically by system wide policies. For example, if durability of an SSD is pursued, moving files that are accessed scarcely and sequentially to an HDD will give us a better life expectancy. In the future, *HetFS* should move files dynamically to appropriate vdevs in response to changes made by the administrator to pursue certain system-wide optimization goals. For instance, *HetFS* could decide to move files that have not been accessed for a certain period of time to a network storage system, which could be represented by another "device" in the ZFS pool. At default, operating system will analyze patterns and will be able to choose between storage media. A file that more than 50% is accessed contiguous will be sent to HDD. If more random access patterns emerge or even byte accesses the file could be sent to a SSD or NVRAM respectively. If a system administrator has created a rule about a file, the automatic decision will not be calculated.

3 Evaluation

This section describes our experiments when testing how several file distributions differently affect the boot process of the Linux kernel. Our experiment platform is a bare metal machine running Ubuntu 16.04 with Linux kernel 4.4.0-21. It is equipped with a processor Intel(R) Core(TM)2 Quad CPU Q9300 @ 2.50GHz with 4 cores. It also has 8GB RAM in 4 modules of 2GB. For storage we have a Seagate BarraCuda at 250GB with 7200rpm and 8MB cache connected with SATA 3.0Gb/s and a Samsung SSD 850 at 250GB with 512MB cache.

3.1 Boot Time

We use *HetFS* to choose in which media to store different boot files in order to see how it affects the booting time of our test machine. We decided to use the boot time because it is a straight forward experiment that heavily involves the underlying file system. Also boot time is crucial when a new system is deployed. Having a simple performance experiment helps us measure if our approach to store files into specific storage media adds a reasonable overhead.

For each experiment, we rebooted the machine 100 times and write the output of systemd-analyze to a file. In Figure 2 there are the boxplots of the median, best and worst total booting times for each run. First we did the experiment with the ZoL [24] version 0.6.5.6 (which is the one that can now be found in the Ubuntu 16.04 repositories), to measure the time of a stable run. Second we run ZoL with 0.7.3-rc3 and commit "935550f" since this is the commit before our code was introduced. This experiment is done in order to see if any major differences have been introduced between the ZFS versions. The third experiment is run by storing only the files that are read during the boot process in the SSD (labeled **RO** in the figure). The fourth experiment, which is labeled **RO+META** in the figure, is a set up where all files that are read during boot time and all ZFS metadata of every file is stored to the SSD. Finally, we add a

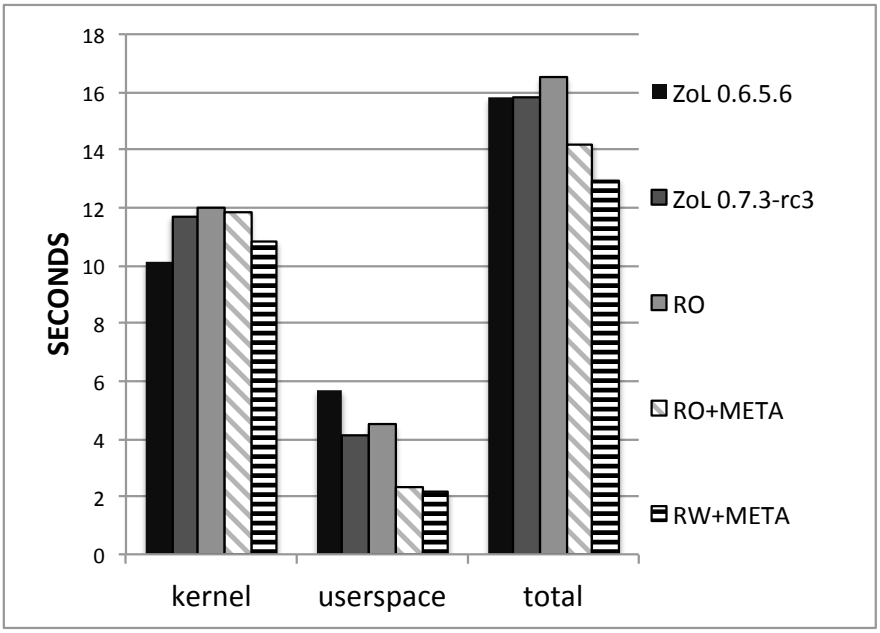

Fig. 1. Mean boot time using different configurations.

	ZFS 0.7.3	RO	RO+META	RW+META
Total size	2MB	1MB	3MB	3.3MB
# of requests	139	28	162	200
Speedup	1x	0.95x	1.11x	1.22x

Table 1. SSD Writes vs. Speedup

fifth experiment where all the files and the metadata are stored in the SSD (labeled **RW+META**). All measurements were done by the systemd-analyze [18] command version 229. The systemd-analyze command returns the time spent in the kernel as well as the time spent in initrd before normal system userspace is reached. A userspace time is also provided which is the time normal system userspace took to initialize.

Figure 1 depicts our results. First of all, we can observe some differences between the 2 versions of ZFS which evidence changes between the versions. For instance, ZFS 0.7 has a 15% performance hit on kernel time but a speed up on userspace time of 27% when compared to ZFS 0.6. Nevertheless, this results in a less than 1% degradation to the total boot time. Placing only the read files in an SSD creates a 2% overhead at kernel time and an 8% overhead to userspace time, which sets the overhead of *HetFS* around 4%, but with a better expected SSD lifetime. In contrast, the results for the fourth run where also the ZFS metadata is stored in the SSD are significantly different: the kernel time

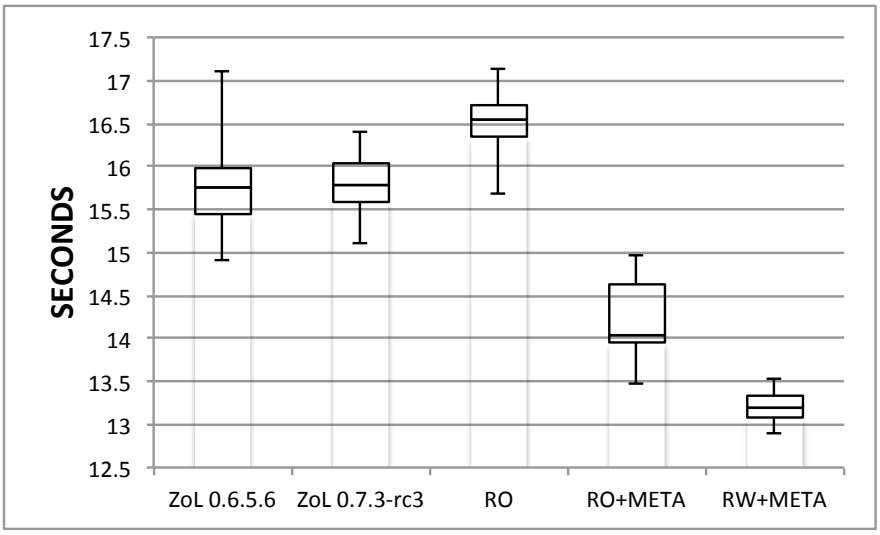

Fig. 2. Median, worst and best boot time for each run.

is almost identical to the ZoL 0.7 baseline, but userspace time yields a 43% speedup. Overall, *HetFS* obtains a final 10% boost, which demonstrates that placing the file system's internal metadata into an SSD can significantly affect performance (and decrease the expected SSD lifetime as well). The final run, where all the data is stored in the SSD, obtains an improvement of a 20% with respect to the plain ZFS, which is to be expected since no data is stored in the slower HDD. Overall the results show that our approach of acquiring the data has a low impact at the responding time of the machine.

3.2 Write Requests

As is apparent from the previous section, significant performance gains can be expected from placing boot files into an SSD. Nevertheless, in order to better understand how much stress the SSD received, we also measured how many I/O requests ended up going to this media, along with the total count and size of the writes operations issued. The results are shown in Table 1. We observe that, since the boot process is not write-intensive, using the SSD to its full capability for storing also the ZFS metadata does not represent a significant load, since only 3MB are requested to be written in the device (**RO+META**). Nevertheless, using the SSD to store exclusively read-only data results in only a 5% drop in performance when compared to a standard ZFS installation, but with significantly less data written to the SSD (1MB/28 requests vs 2MB/139 requests, respectively). Moreover, the worst scenario for an SSD is, as expected, to move all the data to it, which increases the total size of the writes to 3.3MB, but with a 1.22x speedup when compared to standard ZFS.

4 Related Work

Research on hybrid or heterogeneous file systems is divided between how such a system will improve the performance of specific application and designs from scratch. There are numerous examples on specific application optimization, particularly in databases [3, 7–9, 21, 14]. In contrast, our approach treats all applications equally unless the user specifies otherwise.

Hybrid file systems from scratch have their fare share of research. Combo [13] is a Windows-based file system. They achieved file separation because they are looking for free large contiguous parts for storing a file. This approach lacks the ability to automatically change medium based on access patterns. Conquest [20] achieved to mix HDDs and NVMs, but requires special host hardware which ours does not need. A new form of hybrid file system from scratch called N-hybrid was proposed in [10]. N-hybrid utilizes an SSD as a write-through cache for recently used files. Storing a file in a specific medium is possible in N-hybrid but only at the request level. If a file is requested in big chunks, it will be placed in an HDD. Our approach curates files not only by access patterns but also by user needs.

Extending the life of SSDs has also been an issue in recent years. Typically, either an HDD [17] or a NVM-enabled device [15] is used as a caching media at a higher stack level to protect the SSD from writes. Rather than setting up a hierarchy of storage devices, we try to achieve less SSD wearing by statically analyzing the file access patterns, and creating a file distribution that attempts to optimally forward I/O requests to the available devices, instead of just limiting the access to SSD.

Similarly to our work, Oracle has published a white paper [12] where they discuss how to achieve a hybrid storage system within the proprietary ZFS file system. They describe applications that would benefit from this facility but do not discuss any results, performance or otherwise. Instead, our approach focuses in the file system level and how it orchestrates where the files would be stored. Moreover, our work will engulf all storage media and it will make informed decisions based on access patterns on where a file should be placed.

5 Conclusions and Future Work

In this paper, we presented an extension to ZFS aimed to allow system administrators and/or normal users to specify which storage device belonging to a zpool to use to store specific files (e.g. HDDs, SSDs, NVMs or others). Moreover, this extension is presented to the user as a file system with a single mount point. We introduce two separate mechanisms: one for capturing information about file usage, and another that can use this information to decide the placement of individual files. An administrator could use these mechanisms to counteract the drawbacks of one medium with benefits from others, by actively defining which files should be placed in which device.

The experiments done with *HetFS* conclude that the overhead introduced by the mechanisms implemented is low, and different benefits can be achieved

depending on the metric considered. For example, it is possible to reduce nearly 100% the number of writes going to an SSD, which helps with wear-leveling but at the same time somewhat impacts performance, or only move certain number of files to the SSD to have different ratios of performance improvement. Nevertheless, given that different devices have different behaviors, and different I/O workloads have different constraints, these decisions should be taken by automatic mechanisms that can adapt, learn and decide the best placement for a certain target metric.

Thus, the modification presented is a first step to incorporate more advanced or automatic techniques that can take this kind of decisions. Our future research lines are to rely on these mechanisms to automatically detect data access patterns, and define optimization algorithms that are able to use this information to decide the appropriate vdev for a certain file. This algorithms should accurately model a device characteristics and combine this information to target predefined optimization goals (e.g. SSD wear should be reduced by 25% but performance should not drop below 5%). Moreover, placement of file fragments can also be helpful for internally complex files. For example, headers of files that are usually read and written once could be stored in an HDD, whereas the heavily-accessed parts of a database index would benefit from being in NVM. With *HetFS*, we lay the foundation for developing such a system.

6 Acknowledgments

The research leading to these results has received funding from the European Community under the BIGStorage ETN (Project 642963 of the H2020-MSCA-ITN-2014), by the Spanish Ministry of Economy and Competitiveness under the TIN2015-65316 grant and by the Catalan Government under the 2014-SGR-1051 grant. To learn more about the BigStorage project, please visit `http://bigstorage-project.eu/`.

References

1. Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. Skylight—a window on shingled disk operation. *Trans. Storage*, 11(4):16:1–16:28, October 2015.
2. Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 19:1–19:17, New York, NY, USA, 2016. ACM.
3. Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD Bufferpool Extensions for Database Systems. *Proc. VLDB Endow.*, 3(1-2):1435–1446, September 2010.
4. Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83, New York, NY, USA, 2011. ACM.

5. Myoungsoo Jung and Mahmut Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 203–216, New York, NY, USA, 2013. ACM.
6. Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 29:1–29:15, New York, NY, USA, 2016. ACM.
7. Ioannis Koltsidas and Stratis D. Viglas. Flashing Up the Storage Layer. *Proc. VLDB Endow.*, 1(1):514–525, August 2008.
8. Sang-Won Lee, Bongki Moon, and Chanik Park. Advances in Flash Memory SSD Technology for Enterprise Database Applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 863–870, New York, NY, USA, 2009. ACM.
9. Xin Liu and Kenneth Salem. Hybrid Storage Management for Database Systems. *Proc. VLDB Endow.*, 6(8):541–552, June 2013.
10. Jaechun No. NAND Flash Memory-based Hybrid File System for High I/O Performance. *J. Parallel Distrib. Comput.*, 72(12):1680–1695, December 2012.
11. OpenZFS. `http://open-zfs.org/wiki/Main_Page`, January 2017.
12. Oracle. Deploying Hybrid Storage Pools with Oracle Flash Technology and the Oracle Solaris ZFS File System. pages 1–17, Aug 2011.
13. Hannes Payer, MA Sanvido, Zvonimir Z Bandic, and Christoph M Kirsch. Combo drive: Optimizing cost and performance in a heterogeneous storage device. In *First Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, volume 1, pages 1–8, 2009.
14. Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage Management in the NVRAM Era. *Proc. VLDB Endow.*, 7(2):121–132, October 2013.
15. S. Qiu and A. L. Narasimha Reddy. NVMFS: A hybrid file system for improving random write in nand-flash SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, May 2013.
16. Ohad Rodeh and Avi Teperman. zFS : A Scalable Distributed File System Using Object Disks. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, MSS '03, pages 207–, Washington, DC, USA, 2003. IEEE Computer Society.
17. Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD Lifetimes with Disk-based Write Caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.
18. systemd-analyze. `http://manpages.ubuntu.com/manpages/xenial/man1/systemd-analyze.1.html`, January 2017.
19. Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 16:1–16:16, New York, NY, USA, 2016. ACM.
20. An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek. The Conquest File System: Better Performance Through a Disk/persistent-RAM Hybrid Design. *Trans. Storage*, 2(3):309–348, August 2006.
21. Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of*

the 8th ACM International Systems and Storage Conference, SYSTOR '15, pages 6:1–6:11, New York, NY, USA, 2015. ACM.
22. Jiaping Yang, Cheng Peng Henry Tan, and Eng Hong Ong. Thermal analysis of helium-filled enterprise disk drive. *Microsystem technologies*, 16(10):1699–1704, 2010.
23. ZFS Development Team. Rotor vector allocation (small records favour SSD).
24. ZoL: ZFS on Linux. `http://zfsonlinux.org/`, January 2017.