HyCache: A Hybrid User-Level File System with SSD Caching

Dongfang Zhao[⋆] and Ioan Raicu[⋆†]
[⋆]Department of Computer Science, Illinois Institute of Technology, Chicago, IL
[†]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL
dzhao8@hawk.iit.edu, iraicu@cs.iit.edu

I. Introduction

One of the bottlenecks of distributed file systems (DFS), e.g. Google File System [1] and Hadoop Distributed File System [2], is mechanical hard disk drives (HDD): their slow increase in bandwidth, slow decrease in latency, and exponential increase in capacity, have made modern storage devices quite unbalanced. Making things worse, the low bandwidth and high latency of HDD hinders the exploration of data locality, which is critical to distributed computing applications [3]. Even though non-volatile memory e.g. Solid State Drive (SSD), has been introduced for over a decade, HDDs are still dominant storage media in most systems because of their large capacities and low costs.

We propose a middleware called HyCache to manage heterogeneous storage devices for distributed file systems. HyCache provides standard POSIX interfaces through FUSE [4] and works completely in the user space. We show that in the context of file systems, the overhead of user-level APIs (i.e. *libfuse*) is negligible with multithread support on SSD, and with appropriate tuning can even outperform the kernel-level implementation.

II. Design and Implementation

Figure 1 shows a bird's view of HyCache as a middleware between distributed file systems and local storages. At the highest level there are three logical components: request handler, file dispatcher and data manipulator. Request handler interacts with distributed file systems and passes the requests to the file dispatcher. File dispatcher takes file requests from request handler and decides where and how to fetch the data based on some replacement algorithm. Data manipulator manipulates data between two access points of fast- and regular-speed devices, respectively.

The HyCache mount point itself is not only a single local directory but a virtual entry point of two mount points for SSD partition and HDD partition, respectively. Assuming HyCache would be mounted on a local directory called *hycache_mount*, and another local directory (e.g. *hycache_root*) has been created and has at least two subdirectories: the mount point of the SSD partition and the mount point of the HDD partition, users can simply execute *./hycache <root> <mount>* where *hycache* is the executable for HyCache, *root* is the physical directory and *mount* is the virtual directory.

We keep only one single copy of any file at any time to achieve strong consistency. For manipulating files across multiple storage devices we use symbolic links to track file locations. HyCache is implemented for manipulating data at the file level rather than the block level because it is the job of the upper-level distributed file system to chop the big files (e.g. $>$ 1TB). For example in Hadoop Distributed File System, an arbitrarily large file will typically be chunked up in 64MB chunks on each data node.

End users only see virtual files in HyCache mount point (i.e. *hycache_mount*) and every single file in the virtual directory is mapped to the underlying SSD physical directory. SSD only has a limited space so when the usage is beyond some threshold then HyCache will move some file(s) from SSD to HDD and only keep symbolic link(s) to the swapped files. The replacement policy, e.g. LRU or LFU, determines when and how to do the swapping.

HyCache provides two built-in cache algorithms: LRU and LFU. End users are free to plug in other cache algorithms depending on their data patterns and/or application characteristics. We implement LRU and LFU with the standard C library *<search.h>* instead of importing any third-party libraries for queue-handling utilities. This header supports doubly-linked list with only two operation: *insque()* for insertion and *remque()* for removal. We implement all other utilities from scratch e.g. check the queue length, search for a particular element in the queue, etc. Each element of LRU and LFU queues stores some metadata of a particular file like filename, access time, number of access (only useful for LFU though), etc.

HyCache fully supports multithreading to leverage the many-core architecture in most high performance computers. Users have the option to disable this feature to run applications in the single-thread mode. Even though there are cases where multithreading does not help and only introduces overheads by switching contexts, by default multithreading is enabled in HyCache because in most cases this would improve the overall performance by keeping the CPU busy. We will see in the evaluation section how the aggregate throughput is significantly elevated with the help of concurrency.

HyCache has about 2,500 lines of C code, together with some scripts and configuration files. It was compiled with GCC version 4.6.3.

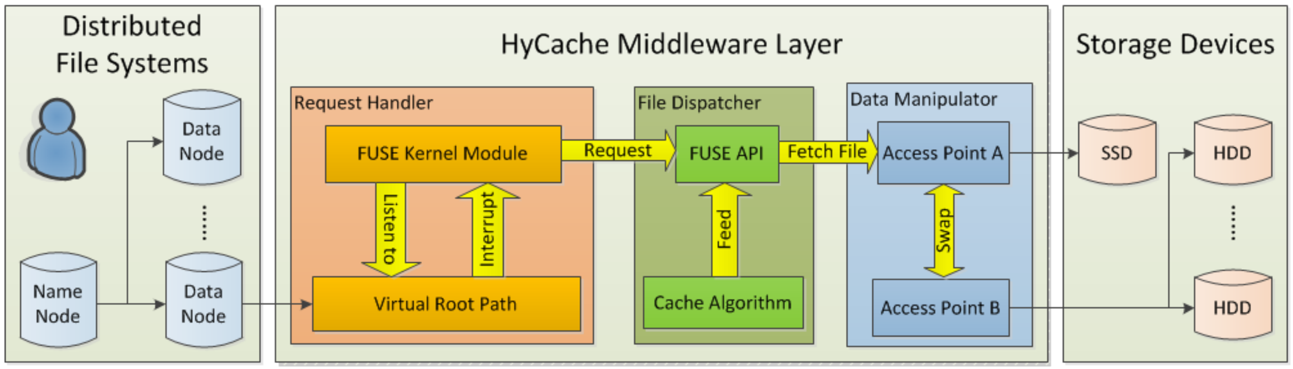
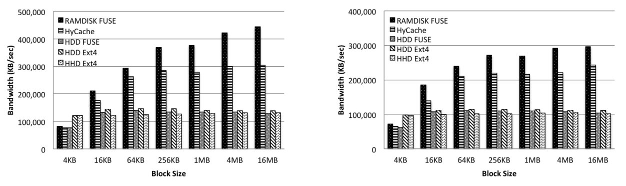


Fig. 1. Three major components in HyCache architecture: Request Handler, File Dispatcher and Data Manipulator.

III. Evaluation

Single-node experiments are carried out on a system comprised of an AMD Phenom II X6 1100T Processor (6 cores at 3.3 GHz) and 16 GB RAM. The spinning disk is Seagate Barracuda 1 TB. The SSD is OCZ RevoDrive2 100 GB. The HHD is Seagate Momentus XT 500 GB (with 4 GB built-in SSD cache). For the experiments on Hadoop the testbed is a 32-node cluster, each of which has two Quad-Core AMD Opteron 2.3GHz processors with 8GB memory. The SSD and HDD are the same as in the single node workstation.

On a single node, we show the throughput with a variety of block sizes ranging from 4 KB to 16 MB. For each block size we show five bandwidths from the left to the right: 1) the theoretical bandwidth upper bound (obtained from RAMDISK), 2) HyCache, 3) a simple FUSE file system accessing a HDD, 4) HDD Ext4 and 5) HHD Ext4. Figure 2(a) shows HyCache read speed is about doubled comparing to the native Ext4 file system for most block sizes. We see a similar result of file writes in Figure 2(b) as file reads.



(a) Read Bandwidth (b) Write Bandwidth

Fig. 2. IOzone bandwidth of 5 file systems.

In the 32-node cluster, we have run two real world applications on HyCache: MySQL and the Hadoop. We install MySQL 5.5.21 with database engine MySIAM, and deploy TPC-H 2.14.3 databases. To test file writes in HyCache, we loaded table *lineitem* at scale 1 (which is about 600 MB) and scale 100 (which is about 6 GB) in these three file systems: LRU HyCache, HDD Ext4 and HHD Ext4. As for file reads we ran Query #1 at scale 1 and scale 100. HyCache has an overall of 9% and 4% improvement over Ext4 on HDD and HHD, respectively. The marginal improvement could be best explained by that the TPC-H is more computation-intensive.

For HDFS [2] we measure the bandwidth by concurrently copying a 1GB file per node from HDFS to the RAMDISK (i.e. $/dev/shm$). The results are reported in Table I, showing that HyCache helps improve HDFS performance by 28% at 32-node scales. We also run the built-in 'sort' example as a real Hadoop application. The 'sort' application is to use mapreduce [5] to sort a 10GB file. We kept all the default settings in the Hadoop package except for the temporary directory which is specified as the HyCache mount point or a local Ext4 directory. The results are reported in Table I.

TABLE I
HDFS Performance

	w/o HyCache	w/ HyCache	Improvement
bandwidth	114 MB/sec	146 MB/sec	28%
sort	2087 sec	1729 sec	16%

IV. Conclusion

In this paper we addressed the long-existing issue with the bottleneck of local spinning hard drives in distributed file systems and proposed a cost-effective solution —HyCache— to alleviate this bottleneck, aimed at delivering comparable performance of an all SSD solution at a fraction of the cost. Our performance evaluation of both micro benchmarks and real applications showed that HyCache can be competitive with kernel-level file systems, and significantly improves the performance of the upper-level distributed file systems.

References

[1] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.
[2] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
[3] Alex Szalay, Julian Bunn, Jim Gray, Ian Foster, and Ioan Raicu. The importance of data locality in distributed computing applications. NSF Workflow Workshop, 2006.
[4] FUSE Project. http://fuse.sourceforge.net.
[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. OSDI, 2004.