A Versatile Persistent Caching Framework for File Systems

Gopalan Sivathanu and Erez Zadok
Stony Brook University

Technical Report FSL-05-05

Abstract

We propose and evaluate an approach for decoupling persistent-cache management from general file system design. Several distributed file systems maintain a persistent cache of data to speed up accesses. Most of these file systems retain complete control over various aspects of cache management, such as granularity of caching, and policies for cache placement and eviction. Hard-coding cache management into the file system often results in sub-optimal performance as the clients of the file system are prevented from exploiting information about their workload in order to tune cache management.

We introduce xCachefs, a framework that allows clients to transparently augment the cache management of the file system and customize the caching policy based on their resources and workload. xCachefs can be used to cache data persistently from any slow file system to a faster file system. It mounts over two underlying file systems, which can be local disk file systems like Ext2 or remote file systems like NFS. xCachefs maintains the same directory structure as in the source file system, so that disconnected reads are possible when the source file system is down.

1 Introduction

Caching is a common way of improving I/O performance. Aside from non-persistent caching, caching of file data from slow, possibly remote, file systems onto faster file systems can significantly increase the performance of slow file systems. Also, the cache remains persistent across reboots. The main problem with most of today's persistent caching mechanisms in file systems like AFS [1] is that the scheme is mostly hardcoded into the file systems such that they provide generic caching policies for a variety of requirements. These generic caching policies often end up being either sub-optimal or unsuitable for clients with specific needs and conditions. For example, in a system where the read pattern is sequential for the entire file, caching it in full during the first read helps in improving performance rather than caching each page separately. The same is not appropriate in a system where reads are random and few. Having the caching mechanism separate helps in integrating persistent caching to any file system. For example, a large IDE RAID array that has an Ext2 file system can cache the recently-accessed data into a smaller but faster SCSI disk to improve performance. The same could be done for a local disk; it can be cached to a faster flash drive.

The second problem with the persistent caching mechanisms of present distributed file systems is that they have a separate name space for the cache. Having the persistent cache directory structure as an exact replica of the source file system is useful even when xCachefs is not mounted. For example, if the cache has the same structure as the source, disconnected reads are possible directly from the cache, as in Coda [2].

In this paper we present a decoupled caching mechanism called xCachefs. With xCachefs, data can be cached from any file system to a faster file system. xCachefs provides several options that can be setup based on user workloads to maximize performance. We implemented a prototype of xCachefs for Linux and evaluated its performance. xCachefs yielded a performance enhancement of 64% and 20% for normal read and read-write workloads respectively over NFS.

2 Background

Persistent caching in file systems is not a new idea. The Solaris CacheFS [5] is capable of caching files from remote file systems to a local disk directory. Solaris CacheFS is tailored to work with NFS as the source file system, though it can be used for other file systems as well. CacheFS does not maintain the directory structure of the source file system in its code. It stores cache data in the form of a database. The Andrew File System [1], a popular distributed file system, performs persistent caching of file data. AFS clients keep pieces of commonly-used files on local disk. The caching mechanism used by AFS is integrated to the file system itself and it cannot be used with any other file system. Coda [2] performs client-side persistent caching to enable disconnected operations. The central idea behind Coda is that caching of data, widely used for performance, can also be exploited to improve availability. Sprite [3] uses a simple distributed mechanism for caching files among a networked collection of workstations. It ensures consistency of the cache even when multiple clients access and update data simultaneously. This caching mechanism is integrated into the Sprite

Network File System.

3 Design

The main goals behind the design of xCachefs are:

- To develop a framework useful to cache file data from any slow file system to a faster file system.
- To provide customizable options that can be set up based on the nature of the workload and access patterns, so as to maximize performance.
- To maintain the directory structure of the cache as the exact replica of the source file system structure, so that disconnected reads are possible without modifying or restarting the applications.

We designed xCachefs as a stackable file system that mounts on top of two underlying file systems, the source and the cache file systems. Stackable file systems incrementally add new features to existing file systems [6]. A stackable file system operates between the virtual file system (VFS) and the lower level file system. It intercepts calls for performing special operations and eventually redirects them to the lower level file systems.

Figure 1 shows the overall structure of xCachefs. During every read, xCachefs checks if the requested page is in the cache file system. On a cache miss, xCachefs reads the page from the source file system and writes it to the cache. All writes are performed on both the cache and source file systems. xCachefs compares the file meta-data during every open to check if the cache is stale and if so it updates or removes the cached copy appropriately. Each check does not require disk I/O as recently-accessed inodes are cached in the VFS icache.

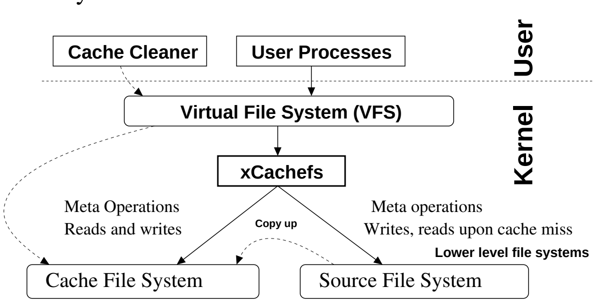

Figure 1: xCachefs Structure

Caching Options. xCachefs caches file data from the source file system either at the page-level or at the file level. It has a configurable parameter called the *size threshold*. Files whose sizes are below the size threshold are cached in full upon an open. Those files that are larger than the size threshold are cached only one page at a time as the pages are read. If files below the size threshold grow due to appends, xCachefs automatically switches to page-level caching for those files. Similarly, if a large file is truncated to a size below the threshold, xCachefs caches it in full the next time it is opened. Based on average file sizes and the access patterns, users can choose a suitable value for the size threshold. Generally it is more efficient to adopt page-level caching for large files that are not accessed sequentially. For small files that are usually read in full, whole file caching is helpful. Since page-level caching has additional overheads due to management of bitmaps, performing page-level caching for small files is less efficient.

Cache Structure and Reclamation. xCachefs maintains in the cache, the same directory structure and file naming as the source. Hence, all file system meta operations like create, unlink, mkdir etc., are performed in an identical manner in both the cache and the source file systems. Maintaining the same directory structure in the cache has several advantages. When the source file system is not accessible due to system or network failures, disconnected reads are possible directly from the cache file system without modifications to the applications that are running. xCachefs has a *disconnected mode* of operation which can be set through an `ioctl` that directs all reads to the cache branch.

Typically, the cache file system has less storage space compared to the source branch. This is because the cache file system is expected to be in a faster media than the source and hence might be expensive. For example, the source file system can be on a large IDE RAID array, and the cache can be a fast SCSI disk of smaller storage. Therefore, we have a user level cleaning utility that runs periodically to delete the least recently-used files from the cache file system, to free up space for new files. The user level cleaner performs cleaning if the cache size has grown larger than an *upper threshold*. It deletes files from the cache until the size is below a *lower threshold*. Typically the upper and lower threshold are around 90% and 70%, respectively.

Implementation. We implemented a prototype of xCachefs as a Linux kernel module. xCachefs has 10,663 lines of kernel code, of which 4,227 lines of code belong to the FiST [7] generated template that we used as our base.

Files whose sizes are smaller than the size threshold (a mount option), are copied in full to the cache branch whenever an `open` for the file is called. Page-level caching is performed in the `readpage` function of the address space operations (`a_ops`). Page bitmaps for partially-cached files are stored in separate files.

Directories are cached during `lookup`, so as to ensure that the directory structure corresponding to a file always exists in the cache branch prior to caching that file. Since a `lookup` is always done before an `open`, the above invariant holds. The `create`, `unlink`, and `mkdir` operations are performed on both cache and source file systems. The `permission` and `readdir` operations are performed only on the source branch.

xCachefs has four `ioctl`s. The STALE_CHECK and NO_STALE_CHECK `ioctls` turn on or off the staleness checking. The CONNECTED and DISCONNECTED `ioctls` switch between connected and disconnected modes of operation.

4 Evaluation

We evaluated the performance of xCachefs with two configurations:

- SCSI-IDE: The source as an Ext2 file system on a 5,200 rpm IDE disk and the cache as an Ext2 file system on a 15,000 rpm SCSI disk.
- SCSI-NFS: The source as an NFS mount over a 100Mbps Ethernet link and the cache branch as an Ext2 file system on a 15,000 rpm SCSI disk.

For all benchmarks we used Linux kernel 2.4.27 running on a 1.7GHz Pentium 4 with 1GB RAM. We unmounted and remounted the file systems before each run of the benchmarks to ensure cold caches. We ran all benchmarks at least ten times and we computed 95% confidence intervals for the mean elapsed, system, user and wait time. In each case, the half widths of the intervals were less than 5% of the mean. Wait time is the elapsed time less CPU time and consists mostly of I/O, but process scheduling can also affect it.

To measure the performance of xCachefs, we used several benchmarks with different CPU and I/O characteristic. For an I/O-intensive benchmark, we used Postmark, a popular file system benchmarking tool. We configured Postmark to create 10,000 files (between 512 bytes and 10KB) and perform 100,000 transactions in 200 directories. For a CPU-intensive benchmark, we compiled the Am-utils package [4]. We used Am-utils version 6.1b3: it contains over 60,000 lines of C code in 430 files. Although the Am-utils compile is CPU-intensive, it contains a fair mix of file system operations. We ran Postmark and Am-utils compilation benchmarks on both SCSI-IDE and SCSI-NFS configurations. We also tested xCachefs using read micro-benchmarks. First, we performed a recursive grep on the kernel source tree. This test was run on both SCSI-IDE and SCSI-NFS setups with warm and cold persistent cache conditions. To evaluate the performance of per page and whole file caching, we wrote a program that reads selected pages from 500 files of size 1MB each. The program has 20 runs; during each run the program reads one page from each of the 500 files. We ran this test with the size threshold values 0 and 409,600 bytes so as to evaluate per page and whole file caching.

Postmark Results. The Postmark results are shown in Figure 2. xCachefs on the SCSI-IDE configuration shows an elapsed time overhead of 44% over the source IDE. This is mostly because the meta operations and writes are done on both source and cache file systems in a synchronous manner. xCachefs on the SCSI-NFS combination shows a small performance improvement of 1.6% elapsed time compared to plain NFS. This is because

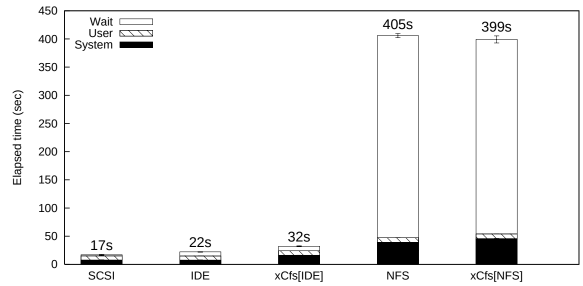

Figure 2: Postmark Results

NFS is several times slower than the SCSI disk which is the cache file system and hence read operations are sped-up by a larger ratio. Postmark is an I/O-intensive benchmark which performs creates, deletes, reads, and writes. Because most of the operations are performed on both file systems, xCachefs does not show a notable performance improvement.

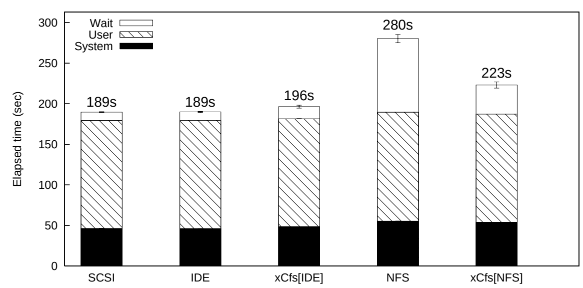

Figure 3: Am-utils Results

Am-utils Results. Figure 3 shows the Am-utils compilation results. xCachefs on the SCSI-IDE combination shows an elapsed time overhead of 3% compared to the source IDE disk. As we can see from the figure, there is no difference in speed between the source IDE and cache SCSI disks for Am-utils compilation. Therefore the overhead in xCachefs is due to the meta-data operations and writes, that are done synchronously to both file systems. On the other hand, for the SCSI-NFS combination, xCachefs showed a performance improvement of 20% elapsed time compared to the source NFS. This is because of the significant speed difference between the source and the cache branches. Therefore to achieve good performance gains from xCachefs for write intensive applications, there must be a significant difference in speeds between the source and cache file systems. For read workloads, the speed difference does not matter much.

Microbenchmarks. Figure 4 shows the grep test results. In the SCSI-IDE configuration, under cold cache conditions, xCachefs showed an overhead of 1.6% elapsed time compared to the source IDE disk. This is because under cold cache conditions, xCachefs performs an additional write to copy data to the cache branch. Under warm cache conditions, xCachefs

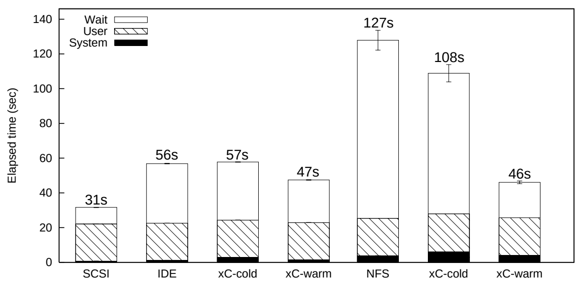

Figure 4: Grep -r test results

showed a performance improvement of 16% elapsed time. This is because all reads are directed to the faster cache branch. In the SCSI-NFS configuration, even under cold cache conditions, xCachefs had a performance improvement of 37%. Here it is interesting to note that even though there is an additional write to the cache file system, xCachefs performs better than the source file system. This is because xCachefs caches small files in full and hence reduces the number of `getattr` calls sent to the NFS server. In plain NFS, computation and I/O are more interleaved during the `grep -r` test, thereby requiring more `getattr` calls, which is not the case with xCachefs over NFS. Under warm cache condition, xCachefs had an improvement of 92% elapsed time, basically because all reads in this case are directed to the faster SCSI disk acting as the cache file system.

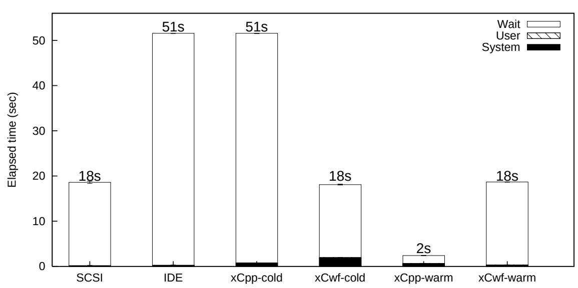

Figure 5: Read micro-benchmark Results

Figure 5 shows the results of the read micro-benchmark under the SCSI-IDE configuration. The results for the SCSI-NFS configuration were mostly similar. This is because this benchmark reads selected pages from files, and thus the total size of data read is less for the network speed to significantly influence the performance. From the figure, we can see that there is a significant difference in speeds of IDE and SCSI disks for this benchmark as it involves a lot of seeks. Under cold cache conditions, the per page caching configuration did not show any difference in performance as the total amount of extra data written to the cache is negligible. However, the whole file caching mode yielded a performance improvement of 64% elapsed time compared to IDE. This is because during the first run, all files are copied to the cache branch and they reside in the memory, eliminating disk requests for subsequent runs. Under warm cache conditions, per-page caching mode yielded a performance enhancement of 95% elapsed time. This is because the pages that are cached in the sparse file are physically close together on disk and hence seeks are reduced to a large extent. The whole file caching mode under warm cache yielded a performance improvement of 63% elapsed time compared to IDE; this is equal to the performance of the cache SCSI disk. This is because all reads are performed only on the cache branch, but there is no seek time reduction in this case.

5 Conclusions

We have described the design and evaluation of an approach for decoupled persistent caching for file systems. Our main goal while designing xCachefs was versatility. File data can be cached from any slow file system to any faster file system while maintaining the original directory and file structure. The IDE-SCSI configuration is a good example for the portability of xCachefs. Our evaluations show that for read intensive applications xCachefs can provide upto 95% performance improvement.

Future Work. We plan to extend xCachefs in order to improve its versatility. We plan to improve performance by performing meta-data operations and file writes asynchronously through a kernel thread. This could also help in parallelizing the I/O in the cache and source file systems. We also plan to support read-write disconnected operation and conflict resolutions.

References

[1] J.H. Howard. An Overview of the Andrew File System. In *Proceedings of the Winter USENIX Technical Conference*, February 1988.

[2] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25, Asilomar Conference Center, Pacific Grove, CA, October 1991. ACM Press.

[3] M.Nelson, B.Welson, and J.Ousterhout. Caching in the sprite network file system. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, 1987.

[4] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[5] SunSoft. Cache file system (CacheFS). Technical report, Sun Microsystems, Inc., February 1994.

[6] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, Raleigh, NC, May 1999.

[7] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000.