FS-Cache: A Network Filesystem Caching Facility

David Howells
Red Hat UK Ltd
dhowells@redhat.com

Abstract

FS-Cache is a kernel facility by which a network filesystem or other service can cache data locally, trading disk space to gain performance improvements for access to slow networks and media. It can be used by any filesystem that wishes to use it, for example AFS, NFS, CIFS, and ISOFS. It can support a variety of backends: different types of cache that have different trade-offs.

FS-Cache is designed to impose as little overhead and as few restrictions as possible on the client network filesystem using it, whilst still providing the essential services.

The presence of a cache indirectly improves performance of the network and the server by reducing the need to go to the network.

1 Overview

The *FS-Cache* facility is intended for use with network filesystems, permitting them to use persistent local storage to cache data and metadata, but it may also be used to cache other sorts of media such as CDs.

The basic principle is that some media are effectively slower than others—either because they are physically slower, or because they must be shared—and so a cache on a faster medium can be used to improve general performance by reducing the amount of traffic to or across the slower media.

Another reason for using a cache is that the slower media may be unreliable for some reason—for example a laptop might lose contact with a wireless network, but the working files might still need to be available. A cache can help with this by storing the working set of data and thus permitting disconnected operation (offline working).

1.1 Organisation

FS-Cache is a thin layer (see Figure 1) in the kernel that permits client filesystems (such as *NFS*, *AFS*, *CIFS*, *ISOFS*) on one side to request caching services without knowing what sort of cache is attached, if any.

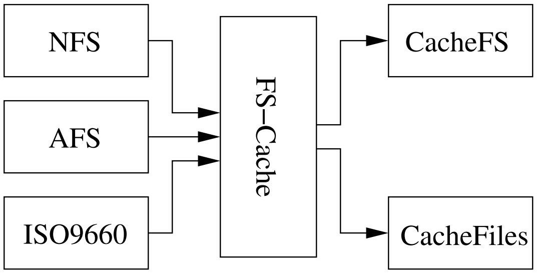

Figure 1: Cache architecture

On the other side *FS-Cache* farms those requests off to the available caches, be they *CacheFS*, *CacheFiles*, or whatever (see section 4)—or the request is gracefully denied if there isn't an available cache.

FS-Cache permits caches to be shared between several different sorts of *netfs*[1], though it does not in any way associate two different views of the same file obtained by two separate means. If a file is read by both *NFS* and *CIFS*, for instance, *two* copies of the file will end up in the cache (see section 1.7).

It is possible to have more than one cache available at one time. In such a case, the available caches have unique tags assigned to them, and a *netfs* may use these to bind a mount to a specific cache.

1.2 Operating Principles

FS-Cache does not itself require that a *netfs* file be completely loaded into the cache before that file may be accessed through the cache. This is because:

1. it must be practical to operate *without* a cache;
2. it must be possible to open a remote file that's *larger* than the cache;
3. the combined size of all open remote files—including mapped libraries—must not be limited to the size of the cache; and
4. the user should not be forced to download an entire file just to do a one-off access of a small portion of it (such as might be done with the `file` program).

FS-Cache makes no use of the `i_mapping` pointer on the *netfs* inode as this would force the filesystems using the cache either to be bimodal[2] in implementation or to always require a cache for operation, with the files completely downloaded before use—none of which is acceptable for filesystems such as *NFS*.

FS-Cache is built instead around the idea that data should be served out of the cache in pages as and when requested by the *netfs* using it. That said, the *netfs* may, if it chooses, download the whole file and install it in the cache before permitting the file to be used—rejecting the file if it won't fit. All *FS-Cache* would see is a reservation (see section 1.4) followed by a stream of pages to entirely fill out that reservation.

Furthermore, *FS-Cache* is built around the principle that the *netfs*'s pages should belong to the *netfs*'s inodes, and so *FS-Cache* reads and writes data directly to or from those pages.

Lastly, files in the cache are accessed by sequences of keys, where keys are arbitrary blobs of binary data. Each key in a sequence is used to perform a lookup in an index to find the next index to consult or, finally, the file to access.

1.3 Facilities Provided

FS-Cache provides the following facilities:

1. More than one cache can be used at once. Caches can be selected explicitly by use of tags.
2. Caches can be added or removed at any time.

[1] Note that the client filesystems will be referred to generically as the *netfs* in this document.

[2] Bimodality would involve having the filesystem operate very differently in each case

3. The *netfs* is provided with an interface that allows either party to withdraw caching facilities from a file (required for point 2). See section 5.

4. The interface to the *netfs* returns as few errors as possible, preferring rather to let the *netfs* remain oblivious. This includes I/O errors within the cache, which are hidden from the *netfs*. See section 5.8.

5. Cookies are used to represent indices, data files and other objects to the *netfs*. See sections 3 and 5.1.

6. Cache absence is handled gracefully; the *netfs* doesn't really need to do anything as the *FS-Cache* functions will just observe a NULL pointer—a negative cookie—and return immediately. See section 5.2.

7. Cache objects can be "retired" upon release. If an object is retired, *FS-Cache* will mark it as obsolete, and the cache backend will delete the object — data and all—and recursively retire all that object's children. See section 5.5.

8. The *netfs* is allowed to propose—dynamically—any index hierarchy it desires, though it must be aware that the index search function is recursive, stack space is limited, and indices can only be children of other indices. See section 3.2.

9. Data I/O is done on a page-by-page basis. Only pages which have been stored in the cache may be retrieved. Unstored pages are passed back to the *netfs* for retrieval from the server. See section 5.7.

10. Data I/O is done directly to and from the *netfs*'s pages. The *netfs* indicates that page A is at index B of the data-file represented by cookie C, and that it should be read or written. The cache backend may or may not start I/O on that page, but if it does, a *netfs* callback will be invoked to indicate completion. The I/O may be either synchronous or asynchronous.

11. A small piece of auxiliary data may be stored with each object. The format and usage of this data is entirely up to the *netfs*. The main purpose is for coherency management.

12. The *netfs* provides a "match" function for index searches. In addition to saying whether or not a match was made, this can also specify that an entry should be updated or deleted. This should make use of auxiliary data to maintain coherency. See section 5.4.

1.4 Disconnected Operation

Disconnected operation (offline working) requires that the set of files required for operation is fully loaded into the cache, so that the *netfs* can provide their contents without having to resort to the network. Not only that, it must be possible for the netfs to save changes into the cache and keep track of them for later synchronisation with the server when the network is once again available.

FS-Cache does not, of itself, provide disconnected operation. That facility is left up to the *netfs* to implement—in particular with regard to synchronisation of modifications with the server.

That said, *FS-Cache does* provide three facilities to make the implementation of such a facility possible: **reservations**, **pinning** and **auxiliary data**.

Reservations permit the *netfs* to reserve a chunk of the cache for a file, so that file can be loaded or expanded up to the specified limit.

Pinning permits the *netfs* to prevent a file from being discarded to make room in the cache for other files. The offline working set must be pinned in the cache to make sure it *will* be there when it's needed. The *netfs* would have to provide a way for the user to nominate the files to be saved, since they, and not the *netfs*, know what their working set will be.

Auxiliary data permits the *netfs* to keep track of a certain amount of writeback control information in the cache. The amount of primary auxiliary data is limited, but more can be made available by adding child objects to a data object to hold the extra information.

To implement potential disconnected operation for a file, the *netfs* must download all the missing bits of a file and load them into the cache in advance of the network going away.

Disconnected operation could also be of use with regard to *ISOFS*: the contents of a CD or DVD could be loaded into the cache for later retrieval without the need for the disc to be in the drive.

1.5 File Attributes

Currently arbitrary file attributes (such as *extended attributes* or *ACLs*) can be retained in the cache in one of two ways: either they can be stored in the auxiliary data (which is restricted in size - see section 1.4) or they can be attached to objects as children of a special object type (see section 3).

Special objects are data objects of a type that isn't one of the two primary types (index and data). How special objects are used is at the discretion of the *netfs* that created it, but special objects behave otherwise exactly like data objects.

Optimisations may be provided later to permit cache file extended attributes to be used to cache file attributes - especially with the possibility of attribute sharing on some backing filesystems. This will improve the performance of attribute-heavy systems such as those that use SE Linux.

1.6 Performance Trade-Offs

The use of a local cache for remote filesystems requires some trade-offs be made in terms of client machine performance:

- **File lookup time**

  This will be INCREASED by checking the cache before resorting to the network and also by making a note of a looked-up object in the cache.
  This should be DECREASED by local caching of metadata.

- **File read time**

  This will be INCREASED by checking the cache before resorting to the network and by copying the data obtained back to the cache.
  This should be DECREASED by local caching of data as a local disk should be quicker to read.

- **File write time**

  This could be DECREASED by doing writeback caching using the disk.
  Write-through caching should be more or less neutral since it's possible to write to both the network and the disk at once.

- **File replacement time**

  This will be INCREASED by having to retire an object or tree of objects from the disk.

The performance of the network and the server are also affected, of course, since the use of a local cache should hopefully reduce network traffic by satisfying from local storage some of the requests that would have otherwise been committed to the network. This may to some extent counter the increases in file lookup time and file read time due to the drag of the cache.

1.7 Cache Aliasing

As previously mentioned, through the interaction of two different methods of retrieving a file (such as *NFS* and *CIFS*), it is possible to end up with two or more copies of a remote file stored locally. This is known as *cache aliasing*.

Cache aliasing is generally considered bad for a number of reasons: it requires extra resources to maintain multiple copies, the copies may become inconsistent, and the process of maintaining consistency may cause the data in the copies to bounce back and forth. It's generally up to the user to avoid cache aliasing in such a situation, though the *netfs* can help by keeping the number of aliases down.

The current *NFS* client can also suffer from cache aliasing with respect to itself. If two mounts are made of different directories on the same server, then two superblocks will be created, each with its own set of inodes. Yet some of the inodes may actually represent the same file on the server, and would thus be aliases. Ways to deal with this are being examined.

FS-Cache deals with the possibility of cache aliasing by refusing multiple acquisitions of the same object (be it an index object or a data object). It is left up to the *netfs* to multiplex objects.

1.8 Direct File Access

Files opened with `O_DIRECT` should not go through the cache. That is up to the *netfs* to implement, and *FS-Cache* shouldn't even see the direct I/O operations.

If a file is opened for direct file access when there's data for that file in the cache, the cache object representing that file should be retired and a new one not created until the file is no longer open for direct access.

1.9 System Administration

Use of the *FS-Cache* facility by a *netfs* does not require anything special on the part of the system administrator, unless the *netfs* designer wills it. For instance, the in-kernel *AFS* filesystem will use it automatically if it's there, whilst the *NFS* filesystem currently requires an extra mount option to be passed to enable caching on that particular mount.

Whilst the exact details are subject to change, it should not be a problem to use the cache with automounted filesystems as there should be no need to wrap the mount call or issue a post-mount enabler.

2 Other Caching Schemes

Some network filesystems that can be used on Linux already have their own caching facilities built into each individually, including *Coda* and *OpenAFS*. In addition, other operating systems have caching facilities, such as *Sun*'s *CacheFS*.

2.1 *Coda*

Coda[1] requires a cache. It fully downloads the target file as part of the open process and stores it in the cache. The *Coda* file operations then redirect the various I/O operations to the equivalents on the cache file, and `i_mapping` is used to handle `mmap()` on a *Coda* file (this is required as *Coda* inodes do not have their own pages). `i_mapping` is not required with FS-Cache as the cache does I/O directly to the *netfs*'s pages, and so `mmap()` can just use the *netfs* inode's pages as normal.

All the changes made to a *Coda* file are stored locally, and the entire file is written back when a file is either flushed on `close()` or `fsync()`.

All this means that *Coda* may not handle a set of files that won't fit in its cache, and *Coda* can't operate *without* a cache. On the other hand, once a file has been downloaded, it operates pretty much at normal disk-file speeds. But imagine running the `file` program on a file of 100MB in size... Probably all that is required is the first page, but *Coda* will download *all* of it—that's fine if the file is then going to be used; but if not, that's a lot of bandwidth wasted.

This does, however, make *Coda* good for doing disconnected operation: you're guaranteed to have to hand the entirety of any file you were working with.

And it does potentially make *Coda* bad at handling sparse files, since *Coda* must download the whole file, holes and all, unless the Coda server can be made to pass on information about the gaps in a file.

2.2 *OpenAFS*

OpenAFS[2] can operate without a cache. It downloads target files piecemeal as the appropriate bits of the file are accessed, and places the bits in the cache if there is one.

No use is made of `i_mapping`, but instead *OpenAFS* inodes own their own pages, and the contents are exchanged with pages in the cache files at appropriate times.

OpenAFS's caching operates using the main model assumed for *FS-Cache*. *OpenAFS*, however, locates its cache files by invoking `iget()` on the cache superblock with the inode number for what it believes to be the cache file inode number as a parameter.

2.3 *Sun*'s *CacheFS*

Modern *Solaris*[3] variants have their own filesystem caching facilities available for use with *NFS* (*CacheFS*). The mounting protocol is such that the cache must manually be attached to each *NFS* mount after the mount has been made.

FS-Cache does things a little differently: the *netfs* declares an interest in using caching facilities when the *netfs* is mounted, and the cache will be automatically attached either immediately if it's already available, or at the point it becomes available.

It would also be possible to get a *netfs* to request caching facilities after it has been mounted, though it might be trickier from an implementation point of view.

3 Objects and Indexing

Part of *FS-Cache* can be viewed as an **object** storage interface. The objects it stores come in two primary types: **index objects** and **data objects**, but other **special object** types may be defined on a per-parent-object basis as well.

Cache objects have certain properties:

- All objects apart from the root index object—which is inaccessible on the *netfs* side of things—have a parent object.

- Any object may have as many child objects as it likes.

- The children of an object do not all have to be of the same type.

- Index objects may only be the children of other index objects.

- Non-index objects[3] may carry data as well as children.

- Non-index objects have a file size set beyond which pages may not be accessed.

- Index objects may not carry data.

- Each object has a key that is part of a keyspace associated with its parent object.

- Child keyspaces from two separate objects do not overlap—so two objects with equivalent binary blobs as their keys but with different parent objects are *different* objects.

- Each object may carry a small blob of *netfs*-specific auxiliary metadata that can be used to manage cache consistency and coherence.

- An object may be pinned in the cache, preventing it from being culled to make space.

- A non-index object may have space reserved in the cache for data, thus guaranteeing a minimum amount of page storage.

Note that special objects behave exactly like data objects, except in two cases: when they're being looked up, the type forms part of the key; and when the cache is being culled, special objects are not automatically culled, but they are still removed upon request or when their parent object goes away.

3.1 Indices

Index objects are very restricted objects as they may only be the children of other indices and they may not carry data. However, they may exist in more than one cache if they don't have any non-index children, and they may be bound to specific caches—which binds all their children to the same cache.

Index object instantiation within any particular cache is deferred until an index further down the branch needs a non-index type child object instantiating within that cache—at which point the full path will be instantiated in one go, right up to the root index if necessary.

Indices are used to speed up file lookup by splitting up the key to a file into a sequence of logical sections, and can also be used to cut down keys that are too long to use in one lump. Indices may also be used define a logical group of objects so that the whole group can be invalidated in one go.

Records for index objects are created in the virtual index tree in memory whether or not a cache is available, so that cache binding information can be stored for when a cache is finally made available.

3.2 Virtual Indexing Tree

FS-Cache maintains a virtual indexing tree in memory for all the active objects it knows about. There's an index object at the root of the tree for *FS-Cache*'s own use. This is the **root index**.

[3] Data objects and special objects

The children of the root index are keyed on the name of the *netfs* that wishes to use the offered caching services. When a *netfs* requests caching services an index object specific to that service will be created if one does not already exist (see Figure 2).

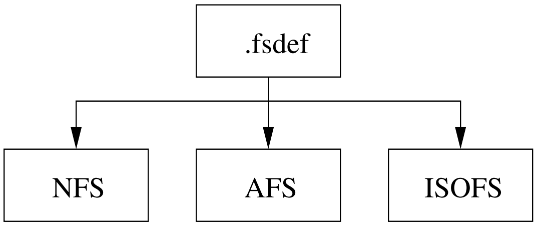

Figure 2: Primary Indices

Each of these is the **primary index** for the named *netfs* , and each can be used by its owner *netfs* in any way it desires. *AFS*, for example, would store per-cell indices in its primary index, using the cell name as the key.

Each primary index is versioned. Should a *netfs* request a primary index of a version other than the one stored in the cache, the entire index subtree rooted at that primary index will be scrapped, and a new primary index will be made.

Note that the index hierarchy maintained by a *netfs* will **not** normally reflect the directory tree that that *netfs* will display to the VFS and the user. Data objects generally are equivalent to inodes, not directory entries, and so hardlink and rename maintenance is not normally a problem for the cache.

For instance, with NFS the primary index might be used to hold an index per server—keyed by IP address—and each server index used to hold a data object per inode—keyed by NFS filehandle (see Figure 3).

The inode objects could then have child objects of their own to represent extended attributes or directory entries (see Figure 4).

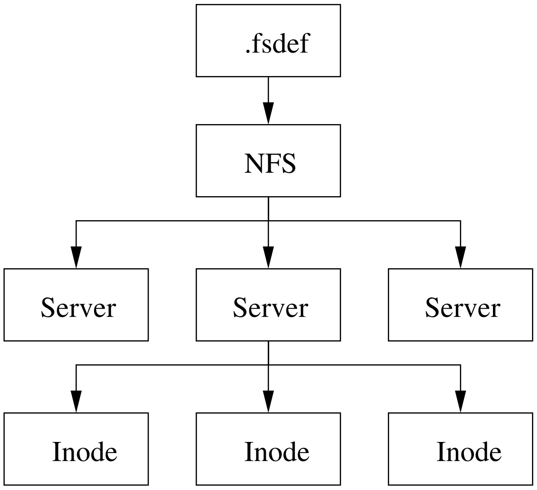

Figure 3: NFS Index Tree

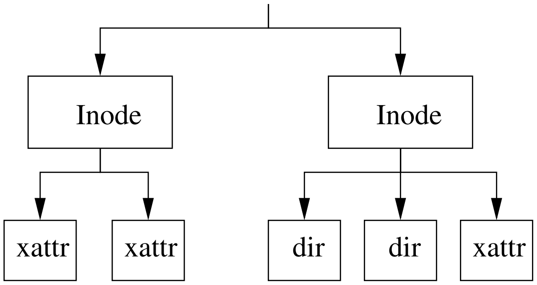

Figure 4: NFS Inode Attributes

Note that the in-memory index hierarchy may not be fully representative of the union of the on-disk trees in all the active caches on a system. *FS-Cache* may discard inactive objects from memory at any time.

3.3 Data-Containing Objects

Any data object may contain quantities of pages of data. These pages are held on behalf of the *netfs*. The pages are accessed by index number rather than by file position, and the object can be viewed as having a sparse array of pages attached to it.

Holes in this array are considered to represent pages as yet unfetched from the *netfs* server, and if *FS-Cache* is asked to retrieve one of these, it will return an appropriate error rather than just returning a block full of zeros.

Special objects may also contain data in exactly the same was as data objects can.

4 Cache Backends

The job of actually storing and retrieving data is the job of a **cache backend**. *FS-Cache* passes the requests from the *netfs* to the appropriate cache backend to actually deal with it.

There are currently two candidate cache backends:

- CacheFS
- CacheFiles

CacheFS is a quasi-filesystem that permits a block device to be mounted and used as a cache. It uses the mount system call to make the cache available, and so doesn't require any special activation interface. The cache can be deactivated simply by unmounting it.

CacheFiles is a cache rooted in a directory in an already mounted filesystem. This is more use where an extra block device is hard to come by, or re-partitioning is undesirable. This uses the VFS/VM filesystem interfaces to get another filesystem (such as *Ext3*) to do the requisite I/O on its behalf.

Both of these are subject to change in the future in their implementation details, and neither are fully complete at the time of writing this paper. See section 6 for information on the state of these components, and section 6.1 for performance data at the time of writing.

5 The *Netfs* Kernel Interface

The *netfs* kernel interface is documented in:

```
Documentation/filesystems/
caching/netfs-api.txt
```

The in-kernel client support can be obtained by including:

```
linux/fscache.h
```

5.1 Cookies

The *netfs* and *FS-Cache* talk to each other by means of **cookies**. These are elements of the virtual indexing tree that *FS-Cache* maintains, but they appear as opaque pointers to the *netfs*. They are of type:

```
struct fscache_cookie *
```

A `NULL` pointer is considered to be a negative cookie and represents an uncached object.

A *netfs* receives a cookie from *FS-Cache* when it registers. This cookie represents the primary index of this *netfs*. A *netfs* can acquire further cookies by asking *FS-Cache* to perform a lookup in an object represented by a cookie it already has.

When a cookie is acquired by a *netfs*, an object definition must be supplied. Object definitions are described using the following structure:

```
struct fscache_object_def
```

This contains the cookie name; the object type; and operations to retrieve the object key and auxiliary data, to validate an object read from disk by it auxiliary data, to select a cache, and to manage *netfs* pages.

Note that a *netfs*'s primary index is defined by *FS-Cache*, and is not subject to change.

5.2 Negative Cookies

A **negative cookie** is a `NULL` cookie pointer. Negative cookies can be used anywhere that non-negative cookies can, but with the effect that the *FS-Cache* header file wrapper functions return an appropriate error as fast as possible.

Note that attempting to acquire a new cookie from a negative cookie will simply result in another negative cookie. Attempting to store or retrieve a page using a negative cookie as the object specifier will simply result in `ENOBUFS` being issued.

FS-Cache will also issue a negative cookie if an error such as `ENOMEM` or `EIO` occurred, a non-index object's parent has no backing cache, the backing cache is being withdrawn from the system, or the backing cache is stopped due to an earlier fatal error.

5.3 Registering The *Netfs*

Before the *netfs* may access any of the caching facilities, it must register itself by calling:

```
fscache_register_netfs()
```

This is passed a pointer to the *netfs* definition.

The *netfs* definition doesn't contain a lot at the moment: just the *netfs*'s name and index structure version number, and a pointer to a table of per-*netfs* operations which is currently empty.

After a successful registration, the primary index pointer in the *netfs* definition will have been filled in with a pointer to the primary index object of the *netfs*.

The registration will fail if it runs out of memory or if there's another *netfs* of the same name already registered.

When a *netfs* has finished with the caching facilities, it should unregister itself by calling:

```
fscache_unregister_netfs()
```

This is also passed a pointer to the *netfs* definition. It will relinquish the primary index cookie automatically.

5.4 Acquiring Cookies

A *netfs* can acquire further cookies by passing a cookie it already has along with an object definition and a private datum to:

```
fscache_acquire_cookie()
```

The cookie passed in represents the object that will be the parent of the new one.

The private datum will be recorded in the cookie (if one is returned) and passed to the various callback operations listed in the object definition.

The cache will invoke those operations in the cookie definition to retrieve the key and the auxiliary data, and to validate the auxiliary data associated with an object stored on disk.

If the object requested is of non-index type, this function will search the cache to which the parent object is bound to see if the object is already present. If a match is found, the owning *netfs* will be asked to validate the object. The validation routine may request that the object be used, updated or discarded.

If a match is not found, an object will be created if sufficient disk space and memory are available, otherwise a negative cookie will be returned.

If the parent object is not bound to a cache, then a negative cookie will be returned.

Cookies may not be acquired twice without being relinquished in between. A *netfs* must itself deal with potential cookie multiplexing and aliasing—such as might happen with multiple mounts off the same *NFS* server.

5.5 Relinquishing Cookies

When a *netfs* no longer needs the object attached to a cookie, it should relinquish the cookie:

`fscache_relinquish_cookie()`

When this is called, the caller may also indicate that they wish the object to be retired permanently—in which case the object and all its children, its children's children, etc. will be deleted from the cache.

Prior to relinquishing a cookie, a *netfs* must have uncached **all** the pages read or allocated to that cookie, and all the child objects acquired on that cookie must have been themselves relinquished.

The primary index should not be relinquished directly. This will be taken care of when the *netfs* definition is unregistered.

5.6 Control Operations

There are a number of *FS-Cache* operations that can be used to control the object attached to a cookie.

`fscache_set_i_size()`
This is used to set the maximum file size on a non-index object. Error `ENOBUFS` will be obtained if an attempt is made to access a page beyond this size. This is provided to allow the cache backend to optimise the on-disk cache to store an object of this size; it does not imply that any storage will be set aside.

`fscache_update_cookie()`
This can be used to demand that the auxiliary data attached to an object be updated from a *netfs*'s own records. The auxiliary data may also be updated at other times, but there's no guarantee of when.

`fscache_pin_cookie()`
`fscache_unpin_cookie()`
These can be used to request that an object be pinned in the cache it currently resides and to unpin a previously pinned cache.

`fscache_reserve_space()`
This can be used to reserve a certain amount of disk space in the cache for a data object to store data in. The reservation will be extended to include for any metadata required to store the reserved data. A reservation may be cancelled by reducing the reservation size to zero.

The pinning and reservation operations may both issue error `ENOBUFS` to indicate that an object is unbacked, and error `ENOSPC` to indicate that there's not enough disk space to set aside some for pinning and reservation.

Both reservation and pinning persist beyond the cookie being released unless the cookie or one of its ancestors in the tree is also retired.

5.7 Data Operations

There are a number of *FS-Cache* operations that can be used to store data in the object attached to a cookie and then to retrieve it again. Note that *FS-Cache* must be informed of the maximum data size of a non-index object before an attempt is made to access pages in that object.

`fscache_alloc_page()`
This is used to indicate to the cache that a *netfs* page will be committed to the cache

at some point, and that any previous contents may be discarded without being read.

`fscache_read_or_alloc_page()`
This is used to request the cache attempt to read the specified page from disk, and otherwise allocate space for it if not present as it will be fetched shortly from the server.

`fscache_read_or_alloc_pages()`
This is used to read or allocate several pages in one go. This is intended to be used from the `readpages` address space operation.

`fscache_write_page()`
This is used to store a netfs page to a previously read or allocated cache page.

`fscache_uncache_page()`
`fscache_uncache_pagevec()`
These are used to release the reference put on a cache page or a set of cache pages by a read or allocate operation.

The allocate, read, and write operations will issue error `ENOBUFS` if the cookie given is negative or if there's no space on disk in the cache to honour the operation. The read operation will issue error `ENODATA` if asked to retrieve data it doesn't have but that it can reserve space for.

The read and write operations may complete asynchronously, and will make use of the supplied callback in all cases where I/O is started to indicate to the *netfs* the success or failure of the operation. If a read operation failed on a page, then the *netfs* will need to go back to the server.

5.8 Error Handling

FS-Cache handles many errors as it can internally and never lets the *netfs* see them, preferring to translate them into negative cookies or `ENOBUFS` as appropriate to the context.

Out-of-memory errors are normally passed back to the *netfs* , which is then expected to deal with them appropriately, possibly by aborting the operation it was trying to do.

I/O errors in a cache are more complex to deal with. If an I/O error happens in a cache, then the cache will be stopped. No more cache transactions will take place, and all further attempts to do cache I/O will be gracefully failed.

If the I/O error happens during cookie acquisition, then a negative cookie will be returned, and all caching operations based on that cookie will simply give further negative cookies or `ENOBUFS`.

If the I/O error happens during the reading of pages from the cache, then if any pages as yet unprocessed will be returned to the caller if the fscache reader function is still in progress; and any pages already committed to the I/O process will either complete normally, or will have their callbacks invoked with an error indication. In the latter case, the *netfs* should fetch the page from the server again.

If the I/O error happens during the writing of pages to the cache, then either the fscache write will fail with `ENOBUFS` or the callback will be invoked with an error. In either case, it can be assumed that the page is not safely written into the cache.

5.9 Data Invalidation And Truncation

FS-Cache does not provide data invalidation and truncation operations per-se. Instead the object should be retired (by relinquishing it with the retirement option set) and acquired anew. Merely shrinking the maximum file size down is not sufficient, especially as representations of extended attributes and suchlike may not be expunged by truncation.

6 Current State

The *FS-Cache* facility and its associated cache backends and *netfs* interfaces are not, at the time of writing, upstream. They are under development at Red Hat at this time. The states of the individual components are as follows:

- *FS-Cache*
  At this time *FS-Cache* is stable. New features may be added, but none are planned.

- *CacheFS*
  CacheFS is currently stalled. Although the performance numbers obtained are initially good, after a cache has been used for a while read-back performance degrades badly due to fragmentation. There are ways planned to ameliorate this, but they require implementation.

- *CacheFiles*
  CacheFiles has been prototyped and is under development at the moment in preference to *CacheFS* as it doesn't require a separate block device to be made available, but can instead run on an already mounted filesystem. Currently only *Ext3* is being used with it.

- *NFS*
  The *NFS* interface is sufficiently complete to give read/write access through the cache. It does, however, suffer from local cache aliasing problems that need sorting out.

- *AFS*
  The *AFS* interfaces is complete as far as the in-kernel *AFS* filesystem is currently able to go. *AFS* does *not* suffer from cache aliasing locally, but the filesystem itself does not yet have write support.

6.1 Current Performance

The caches have been tested with *NFS* to get some idea of the performance. *CacheFiles* was benchmarked on *Ext3* with 1K and 4K block sizes and on also *CacheFS*. The two caches and the block device raw tests were run on the same partition on the client's disk.

The client test machine contains a pair of 200MHz PentiumPro CPUs, 128MB of memory, an Ethernet Pro 100 NIC, and a Fujitsu MPG3204AT 20GB 5400rpm hard disk drive running in MDMA2 mode.

The server machine contains an Athlon64-FX51 with 5GB of RAM, an Ethernet Pro 100 NIC, and a pair of RAID1'd WDC WD2000JD 7200rpm SATA hard disk drives running in UDMA6 mode.

The client is connected through a pair of 100Mbps switches to the server, and the NFS connection was NFS3 over TCP. Before doing each test the files on the server were pulled into the server's pagecache by copying them to `/dev/null`. Each test was run several times, rebooting the client between iterations. The lowest number for each case was taken.

Reading a 100MB file:

Cache	*CacheFiles*		*CacheFS*
state	1K Ext3	4K Ext3	
None	26s	26s	26s
Cold	44s	35s	27s
Warm	19s	14s	11s

Reading 100MB of raw data from the same block device used to host the caches can be done in 11s.

And reading a 200MB file:

Cache	*CacheFiles*		*CacheFS*
state	1K Ext3	4K Ext3	
None	46s	46s	46s
Cold	79s	62s	47s
Warm	37s	29s	23s

Reading 200MB of raw data from the same block device used to host the caches can be done in 22s.

As can be seen, a freshly prepared *CacheFS* gives excellent performance figures, but these numbers don't show the degradation over time for large files.

The performance of *CacheFiles* will degrade over time as the backing filesystem does, if it does—but *CacheFiles*'s biggest problem is that it currently has to bounce the data between the *netfs* pages and the backing filesystems's pages. This means it does a *lot* of page-sized memory to memory copies. It also has to use `bmap` to probe for holes when retrieving pages, something that can be improved by implementing hole detection in the backing filesystem.

The performance of *CacheFiles* could possibly be improved by using direct I/O as well—that way the backing filesystem really would read and write directly from/to the *netfs*'s pages. That would obviate the need for backing pages and would reduce the large memory copies.

Note that *CacheFiles* is still being implemented, so these numbers are very preliminary.

7 Further Information

There's a mailing list available for *FS-Cache* specific discussions:

`mailto:linux-cachefs@redhat.com`

Patches may be obtained from:

`http://people.redhat.com/~dhowells/cachefs/`

and:

`http://people.redhat.com/~steved/cachefs/`

The FS-Cache patches add documentation into the kernel sources here:

`Documentation/filesystems/caching/`

References

[1] Information about *Coda* can be found at:

`http://www.coda.cs.cmu.edu/`

[2] Information about *OpenAFS* can be found at:

`http://www.openafs.org/`

[3] Information about *Sun*'s *CacheFS* facility can be found in their online documentation:

`http://docs.sun.com/`

Solaris 9 12/02 System Administrator Collection » System Administration Guide: Basic Administration » Chapter 40 Using The CacheFS File System (Tasks)

`http://docs.sun.com/app/docs/doc/816-4552/6maoo3121?a=view`

Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Review Committee

Proceedings Formatting Team