Directory-Based Metadata Optimizations for Small Files in PVFS

Bachelorarbeit

Parallele und Verteilte Systeme
Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

Michael Kuhn
Matrikelnummer: 2405219

3. September 2007

Ich versichere, dass ich diese Bachelorarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

.....
Abgabedatum: 3. September 2007

Abstract

In today's file systems each file is made up of data and metadata. The metadata contains some information about the associated data, like ownership and permissions of the file.

While this usually is useful, there are situations when the additional overhead of such a design becomes a problem in terms of performance. This is especially true for cluster file systems, because due to their design every metadata operation is even more expensive. If a user creates several thousand temporary files that are going to be deleted soon anyway, it is not necessary to store detailed information about them.

In this thesis several changes are made to the parallel cluster file system PVFS to better deal with such cases. To do this, PVFS is altered such that certain unnecessary metadata is discarded and therefore metadata performance is increased.

Several tests with a large quantity of files are done to measure the benefits of these changes. The reduction of the metadata overhead halves the time needed for some common file system operations. The speedup of those operations is also analyzed in detail by visualizing the internal workflow of PVFS.

Also, additional work that could be done to further increase the metadata performance as well as possible additions to the actual implementation are presented.

Contents

1 Introduction

In this chapter a definition of file systems is given. It is also described what differentiates parallel and cluster file systems from traditional local file systems. A special focus lies on the metadata maintained by those file systems, because the goal of this thesis is to optimize metadata operations in an already available parallel cluster file system called PVFS[1].

1.1 File Systems

File systems are used to store files that are made up of some data and so-called metadata. The data can be anything, from text to an image or even a video. The metadata contains some information about this data, usually things like:

- the name of the file the data is associated with (file name)
- the owner of the data (ownership)
- who should be allowed to read or write the data (permissions)
- when the file was created, last modified or accessed (timestamps)
- how much data there is (file size)

Physically, the data is usually stored in form of multiple blocks or extents. While blocks mostly have a fixed size and the data is split up to fit into such blocks, extents have a dynamic size and grow as more data should be put into them. Metadata is stored in separate objects, usually called information nodes or inodes. Files are grouped together in directories. Each directory can contain files or other directories, thus creating a directory hierarchy populated with files.

1.1.1 Local File Systems

Local file systems are designed to be used on a single disk partition. Volume managers can be used to group several such partitions together and represent them as one. In any case the whole underlying storage must be accessible by the file system. This means that local file systems are usually constrained to one machine. There are possibilities to span local file systems across several machines, but the details are omitted here.

The file system has access to any data or metadata at any given moment, that is, everything is available on the local disk. Access times for disk access usually lie between 5 ms and 15 ms, dependent on the area of the disk that should be accessed. This is due to the fact that the read/write head of the disk may need to be repositioned. As this is a mechanical operation it takes some time.

[1] Parallel Virtual File System – `http://www.pvfs.org/`

Also, local file systems are usually only accessed by one client at a time. Here, a client is a process that directly modifies the file system. In the case of a local file system this usually is the kernel of the operation system.[2] Normal user programs just request changes through a layer on top of the file system.

1.1.2 Cluster File Systems

Cluster file systems are built in such a way that several machines connected by some network make up one file system. Usually every machine is either a data server or a metadata server, that is, it only stores objects of the respective type. It is also possible but unusual that each machine acts as a combined data and metadata server. PVFS – the cluster file system used in this thesis – employs the former approach.

Every machine only has direct access to its own storage, so no machine has a total overview of the complete file system. So either the servers have to communicate with each other or clients must contact multiple servers. In PVFS each client contacts the servers itself for performance reasons. In any event data has to be sent across the network thus introducing an additional delay between the request and response. For current Ethernet networks this latency usually lies between 50 and 500 μs.

If a client wants to read a file, it first has to figure out on which metadata servers the metadata about this file can be found. Then it must request the data from each data server that holds data of this particular file. In addition to the disk access times of the servers the client has to connect via the network to all those servers. This is expensive due to network latency and bandwidth restrictions.

1.1.3 Parallel File Systems

Parallel file systems can be accessed by multiple remote clients simultaneously. To do this, some form of concurrency control must be in place. This, however, must not be confused with ordinary locking, because it is done above the file system layer. Locking can be used to protect a file against concurrent access, but not the file system itself. Therefore, a mechanism must be implemented at the file system level to keep the clients from corrupting the file system by modifying the same part of it at the same time.

1.2 PVFS

PVFS is a parallel cluster file system, which supports multiple data and metadata servers. The whole file system is made up of several objects, each identified by a unique handle. The concept of objects is described in detail in chapter 3. Each server is responsible for a so-called handle range. Because these handle ranges are non-overlapping, each object is managed by exactly one server. To distribute the load, file data is striped across all available data servers with a default stripe size of 64 KByte. The first data server is chosen randomly, then a round-robin scheme is used.

Figure 1.1 shows an example of this data striping. The top part represents a file of size 352 KByte. At the bottom three data servers are drawn. The arrows show how the file data is striped across all of them.

[2] Or to be more precise, the file system driver within the kernel.

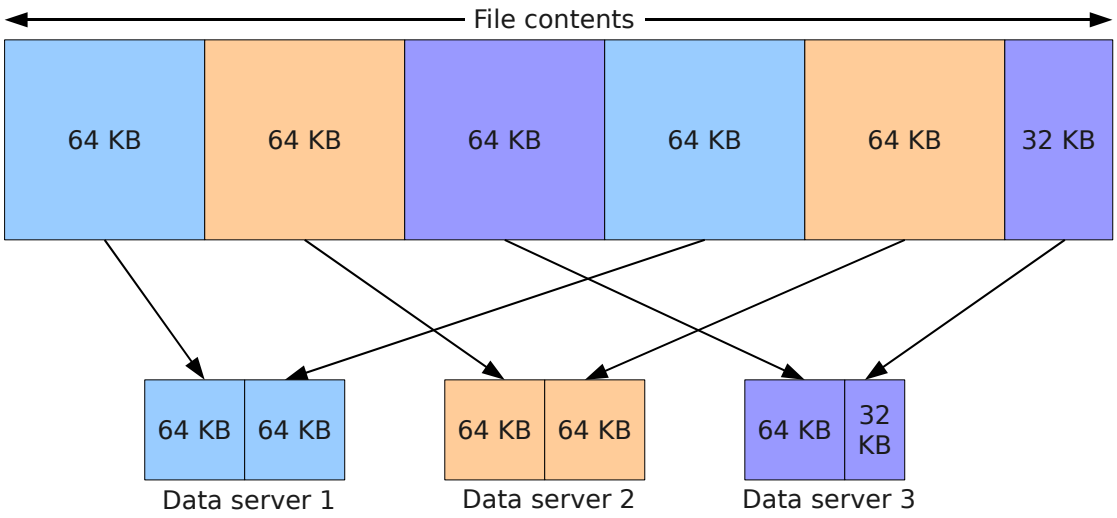


Figure 1.1: Data striping

File metadata however is not distributed. The metadata for any file is managed by exactly one metadata server. Which metadata server is responsible is determined by a hashing algorithm.

PVFS offers several features that make it attractive for research:

- Free software, released under the GPL[3]/LGPL[4]
- The code base has a manageable size (about 200,000 lines of code)
- MPICH2[5] is supported
- Easy to install
- Easy to configure

Summary

In contrast to traditional local file systems, no client or server of a cluster file system has a complete overview of the file system at any point. Therefore communication between them is required. This communication introduces additional cost for each file system operation. The goal of thesis is to reduce this additional cost – at least to some extent – in the parallel cluster file system PVFS, therefore allowing it to perform better in certain scenarios.

[3]GNU General Public License – `http://www.gnu.org/copyleft/gpl.html`
[4]GNU Lesser General Public License – `http://www.gnu.org/copyleft/lgpl.html`
[5]An implementation of the Message Passing Interface – `http://www-unix.mcs.anl.gov/mpi/mpich2/`

2 Motivation

In this chapter a differentiation between data-intensive and metadata-intensive workloads is given. For the sake of simplicity it is assumed that there are no additional limits imposed by the network in terms of bandwidth. Also, some optimization considerations are elaborated.

2.1 Data-Intensive Workloads

If a file system is primarily used to read and write few big files like, for example, multimedia data this is a data-intensive workload. Metadata normally only has to be read or written once for each file and in this case this only takes a fraction of the time needed to read or write the actual data. The bottleneck for this workload is the data servers' disk bandwidth as it determines how fast the data can be read from or written to the disk. An illustration of such data-intensive operations can be found in figure 2.1, where one client reads or writes one large file and thus only has to contact the metadata server once.

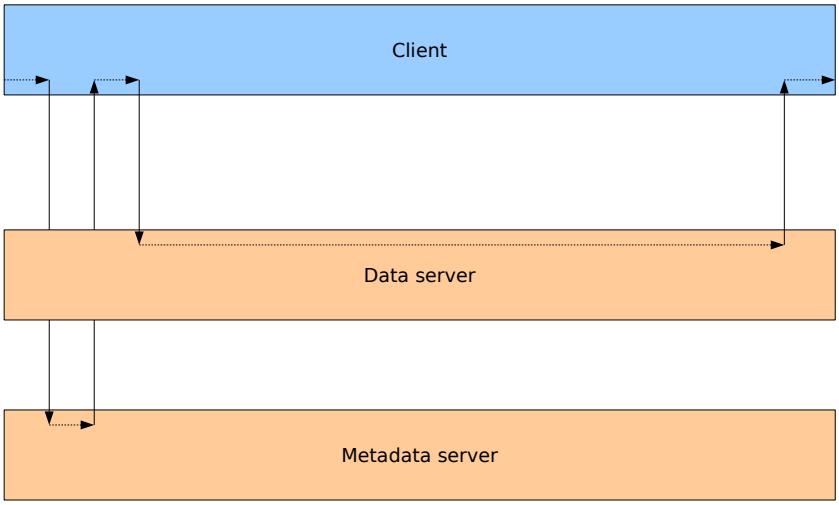


Figure 2.1: Data-Intensive workload

In data-intensive workloads the disk is the bottleneck. They can be characterized as use cases where the majority of data sent across the network is actual file data. Since the file system basically only needs to read or write the data directly to disk, it usually performs at disk speed with such workloads. One approach to increase performance for such workloads – apart from simply using more disks – is to balance the data in such a way that each server's full disk speed can be used. For an example of such an approach's implementation, see [Kun07].

2.2 Metadata-Intensive Workloads

However, if a file system is used to access many small files the workload is considered metadata-intensive. For each access to a file the metadata has to be read or written, so creating, removing or simply listing a huge amount of files in such a file system can put a lot of stress on the metadata servers. Since the metadata is needed to deliver the actual data the number of metadata operations that can be performed in a given time frame also limits the amount of data that can be processed. An illustration of such metadata-intensive operations can be found in figure 2.2, where one client reads or writes many small files and thus has to contact the metadata repeatedly. Since it has to wait for the metadata it can not perform any operations on the data server in the meantime.

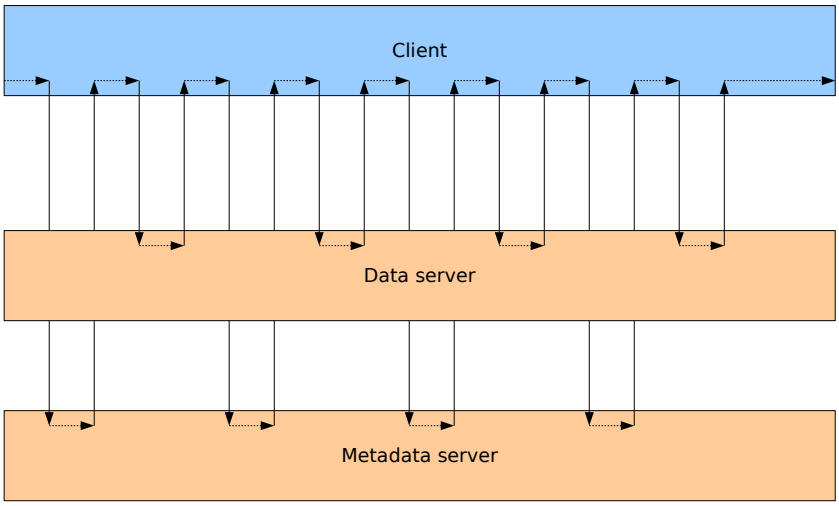


Figure 2.2: Metadata-Intensive workload

With metadata-intensive workloads the metadata throughput is the bottleneck. They can be characterized as use cases where a substantial amount of network traffic is due to metadata. To increase the performance either the time taken for each metadata operation or the total number of metadata operations done has to be decreased. Since the speed of such operations is usually dictated by disk and network latency, the reduction of metadata operations needed for a given task is more feasible.

2.3 Optimization Considerations

There are cases when many small files must be stored in a cluster file system for capacity or speed reasons. If these files are accessed frequently metadata performance plays an important role, therefore a reduction of the number of metadata operations should be considered. Also, if they are only stored temporarily for subsequent processing and deleted afterwards metadata is not really important. There are also cases when metadata must not be stored, because it is either available somewhere else – for example, in a database, maybe even with extended

information – or simply not interesting. This can be used to further increase the performance, because much metadata overhead can be avoided.

2.4 Related Work

Several other approaches can be taken to increase metadata performance, either by focusing on individual file system operations as in this thesis or by trying to improve the overall scalability. One such approach for individual file system operations is presented in [DW07], where only file creation is considered. Multiple strategies to speed up this operation are evaluated. More general approaches are also possible. In [BMLX03] a combination of hashing and caching of parent directory permissions is implemented to reduce the communication overhead, while in [WPBM04] metadata performance is optimized by dynamically partitioning the metadata of the file system tree into subtrees to distribute the load according to the current workload.

2.5 Outlook

The following chapters introduce the actual optimizations done in this thesis. Optimizations are done to three basic file system operations, as shown in chapter 3, that is, only individual file system operations are considered, not the overall metadata design. This chapter also gives an overview of the internal structure of the file system provided by PVFS. Chapter 4 focuses on the actual implementation of these optimizations. Finally, the benefits of these changes are examined in detail. In chapter 5 the actual impact on performance in terms of execution time is evaluated with a relatively simple benchmark program, which simulates parallel accesses. Chapter 6 focuses on the impact of these changes on the low-level operations inside the PVFS server.

Summary

As is described in this chapter, there are fundamental differences between data-intensive and metadata-intensive workloads. Each of these workloads require different approaches to optimize performance. In this thesis a reduction of the number of metadata operations paired with discarding of certain metadata is employed.

3 Design

In this chapter the metadata optimizations' design is explained in an abstract way. To do this, the internal file system structure of PVFS is presented in detail with a special focus on the representation of logical file system objects like files and directories. For a more detailed explanation of the changes necessary to implement these metadata optimizations, see chapter 4.

3.1 Objects

PVFS internally distinguishes several different types of physical objects that can be stored and in turn combined to make up logical objects like files and directories. In order to identify these physical objects each one is assigned a unique handle. The most important physical objects are introduced here. Because there are physical and logical objects with the same name, the physical objects are always identified by the suffix "object" to ease the differentiation. For example, a (logical) directory is made up of a (physical) directory object and a (physical) directory data object.

3.1.1 Metafile Objects

Metafile objects represent logical files. They are used to store file metadata like ownership and permissions, but also all handles of the datafile objects associated with this particular file. The total file size is not stored in the metafile object, but computed dynamically by adding up the respective file sizes of all datafile objects. This is done so that the metafile object does not have to be modified with each operation that changes the size of the file. In turn however, the computation of the file size becomes more expensive with the number of data servers, because in the worst case each one has to be contacted.

Attributes stored for a metafile object:

- POSIX[1] metadata
  - Owner and group
  - Permissions
  - Change, access and modify times
- Datafile distribution, datafile handles and datafile count[2]

[1] Portable Operating System Interface
[2] For an explanation of the distribution, see subsection 3.1.3.

3.1.2 Datafile Objects

Datafile objects are used to store the actual data of files. They are distributed across all data servers. Metadata like ownership and permissions is not stored with each datafile object but rather with the metafile object the datafile object is associated with. This is done because each metafile object can reference multiple datafile objects.

Attributes stored for a datafile object:

- Datafile size (implicitly available through the underlying file system)

3.1.3 Directory Objects

Directory objects represent logical directories. They store directory metadata like ownership and permissions. They also store the handle of a directory data object which in turn stores all files within the directory. So-called directory hints can be set on these directory objects. These hints affect all files within the directory. For example the datafile distribution and the number of datafile objects that should be allocated for a newly created file in this directory can be overwritten.

Attributes stored for a directory object:

- POSIX metadata
  - Owner and group
  - Permissions
  - Change, access and modify times
- Directory entry count
- Directory hints
  - Distribution name and parameters (`dist_name` and `dist_params`)
  - Datafile count (`num_dfiles`)

Directory Hints

The directory hints are currently mostly used to control the distribution of file data across the data servers. `dist_name` and `dist_params` are used to automatically set a distribution function for every new file. Distribution functions control the way file data is striped. For example, one data server could receive twice the amount of data all other data servers receive. This could be used to balance the load if servers of different capacity are used. The `num_dfiles` hint is simply used to assign the number of datafile objects that should be used for a file. Normally – if the file is big enough – one datafile object is created on each data server. This hint can be used to, for example, force that a file is striped only across 2 data servers. However, directory hints can be used to influence other behavior concerning the files created within the directory the directory hint is set on. For example, a new directory hint is used to give the user a possibility to turn the metadata optimizations on and off.

3.1.4 Directory Data Objects

Directory data objects store pairs of the form `file_name: metafile_handle` to identify all files within the directory the directory data object is associated with. This indicates that the file represented by the metafile object with the handle `metafile_handle` is available as the file called `file_name` within this particular directory. Further information is not stored, because it is already available from the associated directory object. There exists a one-to-one mapping between directory objects and directory data objects, that is, each directory object references exactly one directory data object and each directory data object is associated with exactly one directory object. This separation is done transparently to the client. If a client requests all directory entries, both objects are read by the server and returned as one.

Attributes stored for directory data object:

- Directory entries

3.1.5 Example Directory Tree

To illustrate the usage of the presented objects, an example is given in figure 3.1. It shows all objects and relations between them necessary to build a simple directory tree with only a single directory that contains one file. The figure represents the current implementation in PVFS. A directory called `testdir` exists in the file system's root directory (`/`) and contains a file called `testfile`. Consequently, this file is accessible as `/testdir/testfile`.

The numbers behind the object type represent the unique object handles.[3] Other information within the boxes stands for metadata set on the respective object. "UID" and "GID" are abbreviations for user ID and group ID respectively, two pieces of common metadata that can be set on any object. "Permissions" stands for the octal representation of the file access permissions. For example, a value of `0644` means that anyone may read the file, but only the owner may write it.[4] It is noteworthy that this information is not set on the datafile objects at all, but only on the associated metafile object.

The directory objects and directory data objects are represented as one, because from the client's point of view there is only a directory object. Everything that has to do with the directory data object is done transparently by the server. Also, differentiating between them would only complicate matters.

As can be seen, this is quite a complex structure for a simple directory that contains a single file. It is obvious that this structure becomes much more complex when a whole directory hierarchy contains thousands of files that are striped across multiple servers. The goal of this thesis is to reduce this complexity for certain use cases. Multiple datafile objects per file and the indirections caused by the metafile object make up a huge part of this complexity. Therefore, a feasible approach is to consolidate the metafile object and its datafile objects somehow.

[3] The handles are made up and not taken from a real world example.
[4] See `man chmod` for more information.

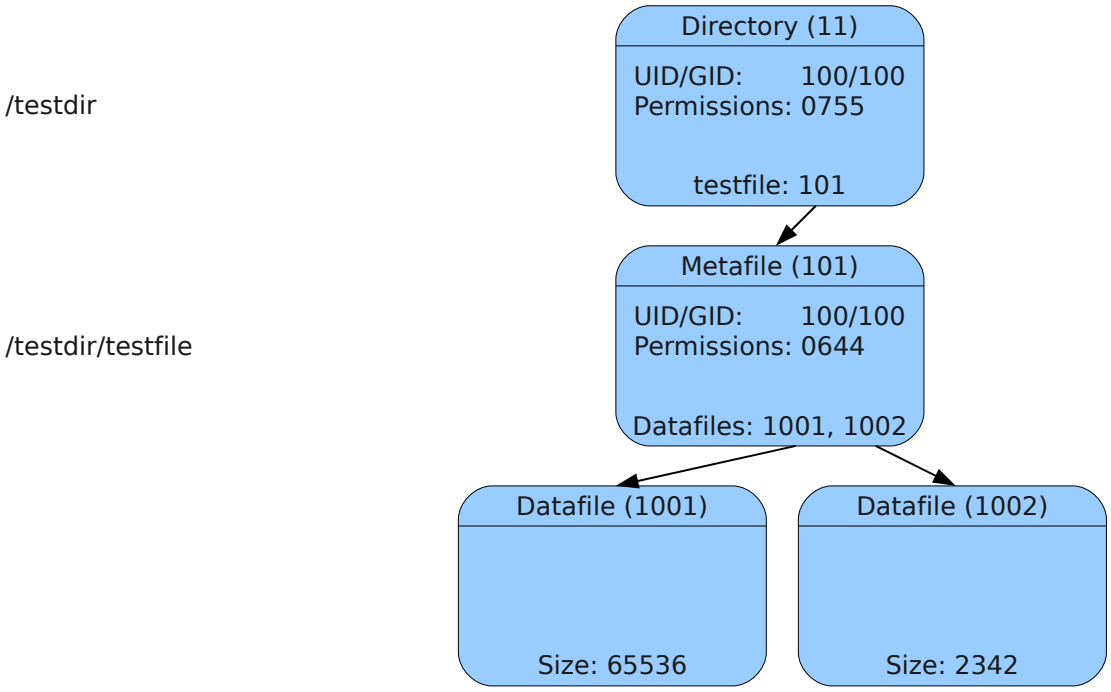


Figure 3.1: Normal directory tree

3.2 Metadata Optimizations

The example in figure 3.1 is now used to illustrate the desired metadata optimizations introduced in this thesis. These metadata optimizations are implemented as a new directory hint, that is, they can be turned on and off on a per-directory basis.

The metafile object's purpose is to link together all datafile objects that belong to a particular file. It is obvious that the metafile object can be omitted if only one datafile object exists. For small files it is not really necessary to create multiple datafile objects, so in this particular use case the need for a metafile object can be bypassed. If only one datafile object is created for each file, the datafile object's handle can simply be put into the directory data object's list of directory entries.

Figure 3.2 shows the same file system tree as figure 3.1, but this time with metadata optimizations applied. In particular, there now is only one datafile object and no metafile object. As can be seen, these metadata optimizations affect both the actual file and the directory in which it is located. Instead of a metafile object that references several datafile objects there now is only one datafile object that stores all file data. Also, the datafile object's handle is used instead of the metafile object's handle to reference the file in the list of directory entries. It is also worth mentioning that no common metadata is set on the datafile object at all. Common metadata like ownership and permissions could be set on the datafile object itself, since this metadata can be set on every object. This is not done for performance reasons, because another message

would need to be sent to the appropriate data server to retrieve this information. Setting the common metadata on the datafile object would, however, make the faking of this metadata unnecessary, as can be seen in chapter 4.

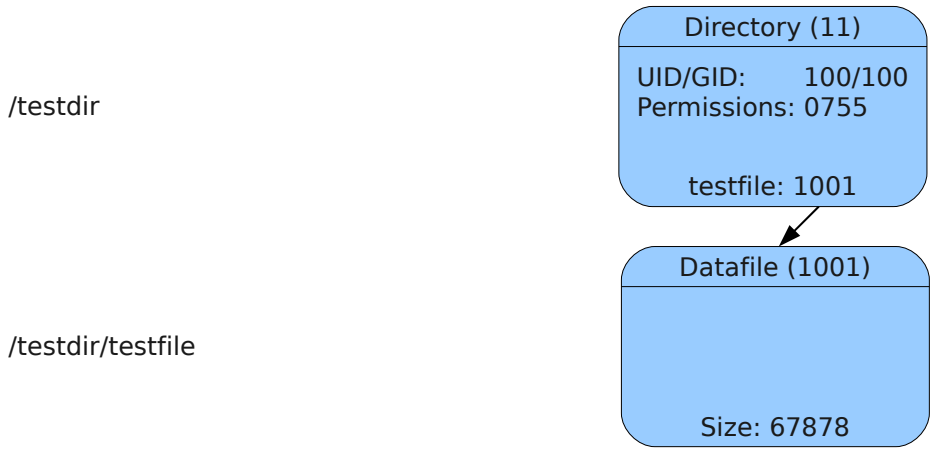


Figure 3.2: Optimized directory tree

With these changes made, however, several problems have to be considered:

1. The limit of one datafile object per file must be enforced, otherwise the file system ends up corrupt, because the other datafile objects would not be referenced by any metafile object or directory entry and therefore be lost.

2. The client and server expect a metafile object to be present. This metafile object stores all metadata of a file, so this information must be faked in some way.

On the other hand, the following advantages become apparent:

1. No metadata server has to be contacted if a file needs to be read or written.

2. Only one data server needs to be contacted for each file. Additionally, the total file size is available directly, avoiding expensive computation. This even applies to small files, since the default striping size is only 64 KByte.

3.2.1 Drawbacks

On the other side, this has impact on the file system semantics, because certain metadata is not stored at all anymore. However, since the metadata optimizations are implemented as a directory hint, users must explicitly enable them and therefore should know what to expect. Consequently, if these metadata optimizations are not activated, they do not influence the normal operation of PVFS in any way. Also, file data is now only stored on one data server,

which may decrease performance for larger files. Since the metadata optimizations are to be used with small files, however, this is to be expected. In theory, if the metadata optimizations are enabled for some files, it could be possible for other users to access and modify these files, because no ownership information and permissions are available, thus rendering permission checks useless. Tests with PVFS's administration tools indicate that this is at least not possible with these tools. However, the file system can also be accessed by other means. Currently, it is not known whether these could be used to circumvent access restrictions.

3.2.2 Alternative Optimizations

In this thesis two optimization approaches are combined. On the one hand, a change of the file system semantics is done by discarding certain metadata. In particular, no POSIX metadata – that is, ownership information, permissions and timestamps – is available for files that are created with the metadata optimizations applied. On the other hand, due to the elimination of the metafile object, less work is needed. This is explained in detail in section 3.3 and chapter 4.

Other possible optimization approaches include client-side caching, relaxed consistency requirements and so-called compound operations. Most of these approaches are already implemented to some extent. Consequently, their impact on metadata performance is mostly known and therefore they are only presented here briefly.

Client-Side Caching

At the moment client-side caching is used only to store metadata for read operations. Modifying metadata operations are not cached at all and consequently, each modification requires communication with the metadata server. Details about this can be found in chapter 4. This is closely related with the consistency requirements of PVFS, because such caching of metadata could cause the clients' view of the file system to be inconsistent. However, it could also dramatically speed up certain use cases. For example, the parallel cluster file system Lustre[5] plans to support metadata caching for write operations in the future.

Relaxed Consistency Requirements

Each file system operation is immediately executed on the server such that each client can see the changes and therefore the clients' view of the file system is consistent. For example, if all files in a directory are changed, that means that for every file several messages have to be sent to the server. It would also be possible to combine all changes into one batch operation and then send this to the server once. This would decrease communication overhead considerably, but also have an impact on file system consistency.

Compound Operations

Currently, file system operations access the same object several times. For example, if a new file is created, a new metafile object is created and then later its attributes are set, as can be seen in section 3.3. This could be combined into a single operation, such that the server has not to be contacted multiple times. An implementation of this is presented in [DW07].

[5] For more information about Lustre, see `http://www.lustre.org/` and `http://www.clusterfs.com/`.

3.3 File System Operations

Three basic file system operations are adapted to make use of these optimizations. Each of these operations can be split into several smaller steps that are executed consecutively. A reduction of the number of these steps increases performance, therefore it is now analyzed which of these steps can be skipped safely. The actual implementation of these operations and changes is described in detail in chapter 4.

Even though only these three file system operations are adapted and examined here, all other common file system operations – like copying or moving a file – work, too. However, these three are best suited to demonstrate the metadata optimizations, because other file system operations include additional overhead. For example, when copying a file, obviously the actual file data has to be transferred as well.

3.3.1 File Creation

The following steps are necessary to create a new file in a directory:

1. Get the directory's attributes
2. Create the metafile object
3. Create the datafile objects
4. Set the metafile object's attributes
5. Create a directory entry for the file

To implement the metadata optimizations steps two and four are skipped. Also, it is enforced that only one datafile object is created. The handle of this single datafile object is used instead of the metafile object's handle for the directory entry.

3.3.2 File Listing

The following steps are necessary to list the metadata of a file:

1. Get the metafile object's attributes
2. Get the file size of each datafile object

Since there is no metafile object anymore, step one is skipped. Also, only one datafile object exists and therefore only one data server has to be contacted to request the file size. The metadata usually stored as the metafile object's attributes is faked.

3.3.3 File Removal

The following steps are necessary to remove a file from a directory:

1. Remove the file's directory entry
2. Get the metafile object's attributes

3. Remove the datafile objects
4. Remove the metafile object

Again, as there is no metafile object step four is skipped. Step two can not be skipped, because it is needed to determine if a file was created with the `no_metafile` hint set or not.

Summary

As shown in this chapter, the elimination of the metafile object offers several opportunities for performance improvement within the specific file system operations. In the course of this thesis the following three basic operations are inspected closely and adapted to take advantage of the changes described in this chapter:

1. File creation – creating all necessary data structures to represent a file
2. File listing – requesting and showing all metadata available for a given file
3. File removal – destroying all data structures associated with a file

Chapter 4 gives an in-depth look at the changes made to the PVFS source code to implement the metadata optimizations. The actual improvements of these changes are also evaluated by using benchmarks and visualization in chapter 5 and chapter 6.

4 Implementation

To implement the changes outlined in chapter 3 several changes to PVFS are necessary. They are explained in detail in this chapter, as well as some fundamental PVFS internals. An important goal is to keep the changes minimal to ease future development. All these changes are implemented on top of the modified version of PVFS that is also used in [Kun07]. However, a port to the official version of PVFS should be relatively easy.

4.1 Internals

4.1.1 Architecture

PVFS uses the layered architecture shown in figure 4.1.[1] It is split into a client and a server side, each made up of several layers. These layers abstract the functionality provided by the layers beneath them. All communication between the layers is done in a non-blocking way, that is, all functions called return immediately and can later be checked for completion.

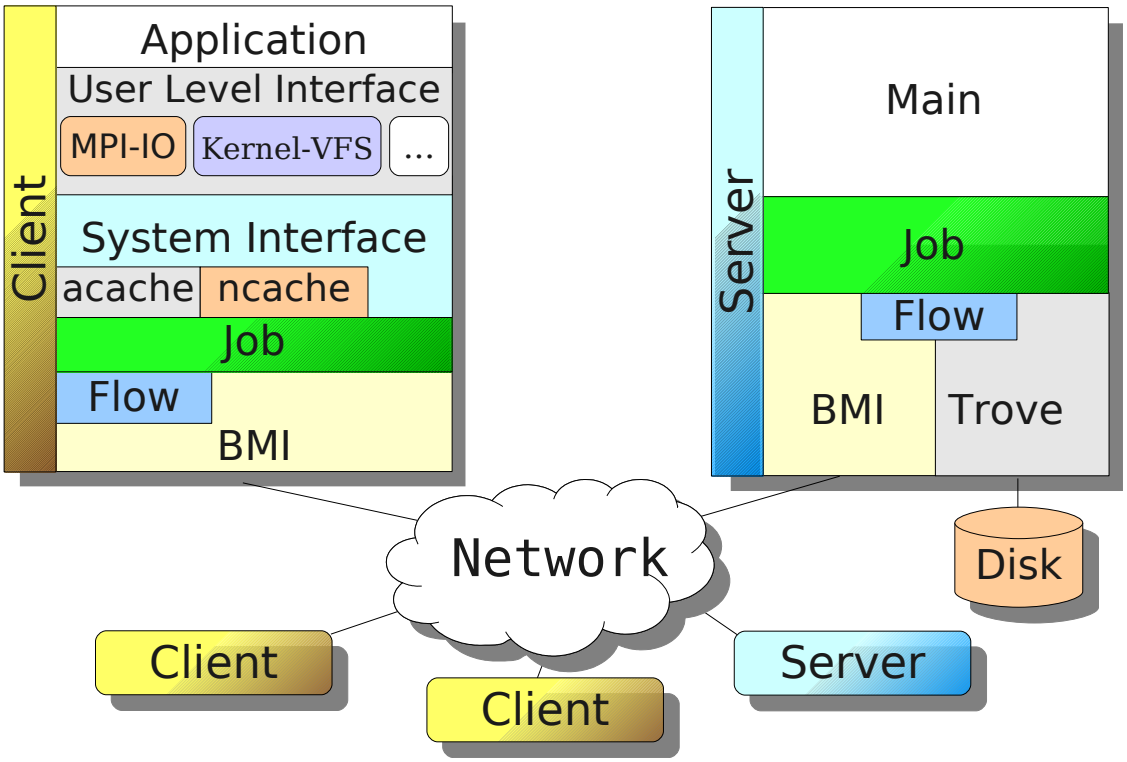


Figure 4.1: PVFS's layered architecture

[1]The figure is taken from [Kun07] with the author's permission.

This split and the most important of these layers are now described. For more in-depth information about this architecture, see [Kun07] and the documentation in the PVFS source package.

Server

Each instance of the server side – that is, each server process – can either act as a data server, a metadata server or both. Usually only one of these server processes is started on a machine and several of them make up a file system, thus distributing the file system across several machines, as described in chapter 1. The server runs three threads to parallelize the work:

- One thread manages all BMI communication.
- One thread manages all Trove I/O.
- The main thread handles all the remaining work.

Client

A client is a process that accesses and possibly manipulates a file system provided by the server processes. Each client has direct access to the file system, either through one of the high-level "User Level Interfaces" or by directly using the "System Interface".

BMI

The Buffered Message Interface provides a network-independent interface for both the clients and the servers to communicate with each other. BMI currently supports TCP, Myrinet and InfiniBand.

Trove

The Trove layer handles everything related to actual I/O[2] to and respectively from the underlying persistent storage. This layer also provides a storage-independent interface. Currently only one module is available, which writes data to files in an underlying local file system and metadata to a Berkeley database. For example, this independence can be used to eliminate disk latency by using the RAM instead of the disk and thus ease profiling. For more information on this particular use case, see [Kun06].

System Interface

All changes made in this thesis mainly take place in this layer. This has the advantage that all higher layers automatically benefit from the changes. This layer provides an API[3] and a corresponding library called `libpvfs` that is used by the "User Level Interface". On this layer all clients access the file system independently. If desired, advanced implementations can be done in the "User Level Interface". Direct use of the `libpvfs` library should be avoided, because it provides only a very low-level interface and is quite tedious to use.

[2] Input/Output
[3] Application Programming Interface – Basically a set of functions abstracting the provided functionality.

User Level Interface

This layer is used to abstract access to the file system even further by extending the functionality provided by the `libpvfs` library. There are currently two commonly used implementations of this layer: a module for ROMIO and a kernel module for Linux. ROMIO is the I/O component of MPICH2 and provides a high-level interface for parallel I/O with, for example, collective file system operations. The kernel module on the other hand enables PVFS to be accessed via the POSIX API for I/O.

acache

The attribute cache stores the metadata of files and directories for a limited amount of time. Consequently, the server does not need to be contacted repeatedly if the metadata of a specific file or directory is needed multiple times.

ncache

The name cache stores mappings between file names and their corresponding object handles. Consequently, a file name does not need to be resolved repeatedly if a file or directory is accessed multiple times.

4.1.2 State Machines

In PVFS file system operations are implemented as state machines. These state machines consist of several states and transitions between them. A given state machine is in only one state at a time. In each state either a specific function or another so-called nested state machine is executed. For example, the operations and individual steps in section 3.3 are represented by state machines and their states respectively.

Both can return a value to the calling state machine that determines which transition will be taken. This concept is elaborated with the example client state machine in listing 4.1. Server state machines work slightly differently, but are omitted here, because the changes in this chapter are mainly done to client state machines.

Listing 4.1: Example client state machine

```
1  machine my_state_machine
2  {
3      state init
4      {
5          run my_sm_init;
6          success => cleanup;
7          default => some_state;
8      }
9
10     state some_state
11     {
12         jump my_other_state_machine;
13         default => cleanup;
```

```
14      }
15
16      state cleanup
17      {
18          run my_sm_cleanup;
19          default => terminate;
20      }
21  }
```

The state machine's name is given after the `machine` keyword. In case of a nested state machine, `state` would be preceded by `nested`. A set of named states follows, each introduced by `state`. The first line specifies which function or nested state machine is executed in this particular state. The `run` statement executes a function, while `jump` executes a nested state machine. For each state several transitions of the form `return_value => new_state` can be given. If no return value matches, the `default` transition is used. The `default` transition must be given last, because the first matching transition is used.
The state machine begins in the `init` state and ends if it reaches the `terminate` state.

Functions run in each state are defined as in listing 4.2.

Listing 4.2: Example client state function for initialization

```
1  static int my_sm_init (PINT_client_sm *sm_p, job_status_s *js_p)
2  {
3      js_p->error_code = 0;
4
5      return 1;
6  }
```

The return value that defines the transition is stored in `js_p->error_code`, with `success` being an alias for `0`. The return value of the function itself determines the next action of the state machine. If `1` is returned, the transition to the next state is taken, while a `0` signals that the state machine should be terminated.

Listing 4.3: Example client state function for cleanup

```
1  static int my_sm_cleanup (PINT_client_sm *sm_p, job_status_s *js_p)
2  {
3      js_p->op_complete = 1;
4
5      return 0;
6  }
```

In the example from listing 4.1 the `init` state calls the `my_sm_init` function, which in turns sets `js_p->error_code` to `0` – which equals `success`. Therefore the transition to `some_state` is used. This state calls a nested state machine called `my_other_state_machine`. After the nested state machine completes, the transition to the `cleanup` state is used regardless of the return

value of `my_other_state_machine`. The function `my_sm_cleanup` from listing 4.3 is called. It sets `js_p->op_complete` to `1`, indicating that this state machine is finished. It then returns `0`, causing the state machine to be terminated.

4.1.3 Message Pairs

The PVFS client communicates with the server via so-called message pairs, that is, a request sent by the client and a response sent back by the server. These message pairs are handled by a special nested state machine called `pvfs2_msgpairarray_sm` that is used both by the client and the server. On the client this state machine sends a request and waits for a response from the server. In case an error occurs, the request is sent again. For instance this might happen if the server crashes during the processing of the request.

4.2 New Directory Hint: `no_metafile`

To be able to control the metadata optimizations per directory a so-called directory hint is added. This directory hint can be set on any directory and influences files and directories created inside it. The directory hint is called `no_metafile` and is available as the extended attribute `user.pvfs2.no_metafile`. Without this directory hint being set PVFS behaves as if no changes were made.

To introduce this new directory hint changes to several areas of PVFS must be made. There is some common infrastructure that needs to be modified to support a new directory hint. Also, the directory hint must be introduced to the client and the server, as well as the Request Protocol. Finally, it is also shown how the hint can be used.

4.2.1 Common Infrastructure Modifications

To introduce a new directory hint it has to be added at several places. First, the name of the new hint must be defined in `src/common/misc/pvfs2-internal.h` which contains all directory hints interpreted by PVFS. An excerpt from this file is shown in listing 4.4.

Listing 4.4: Excerpt from `src/common/misc/pvfs2-internal.h`

```
1  ...
2
3  /* Optional xattrs have "user.pvfs2." as a prefix */
4  #define SPECIAL_DIST_NAME_KEYSTR          "dist_name\0"
5  #define SPECIAL_DIST_NAME_KEYLEN           21
6  #define SPECIAL_DIST_PARAMS_KEYSTR        "dist_params\0"
7  #define SPECIAL_DIST_PARAMS_KEYLEN         23
8  #define SPECIAL_NUM_DFILES_KEYSTR         "num_dfiles\0"
9  #define SPECIAL_NUM_DFILES_KEYLEN          22
10 #define SPECIAL_METAFILE_HINT_KEYSTR      "meta_hint\0"
11 #define SPECIAL_METAFILE_HINT_KEYLEN      21
12 #define SPECIAL_NO_METAFILE_KEYSTR        "no_metafile\0"
13 #define SPECIAL_NO_METAFILE_KEYLEN         23
14
```

```
15 ...
```

The key string has to be manually terminated with \0. The key length can be computed by concatenating the prefix and the key string including the terminating \0, in this case user.pvfs2.no_metafile\0, which has length 23.

For an explanation of the other directory hints, see chapter 3.

4.2.2 Request Protocol Modifications

The hint then has to be added to the PVFS_directory_hint structure that stores all possible directory hints for a given directory. This structure can be found in the file src/proto/pvfs2-attr.h. An excerpt is given in listing 4.5.

Listing 4.5: Excerpt from src/proto/pvfs2-attr.h

```
1  /* extended hint attributes for a directory object */
2  struct PVFS_directory_hint_s
3  {
4      uint32_t  dist_name_len;
5      /* what is the distribution name? */
6      char      *dist_name;
7      /* what are the distribution parameters? */
8      uint32_t  dist_params_len;
9      char      *dist_params;
10     /* how many dfiles ought to be used */
11     uint32_t dfile_count;
12     /*
13      * If this hint is set to any value not equal to 0,
14      * exactly one datafile and no metafile will be created
15      * for new files in this directory.
16      */
17     uint32_t no_metafile;
18 };
19 typedef struct PVFS_directory_hint_s PVFS_directory_hint;
```

The PVFS_directory_hint structure is actually part of a larger structure called PVFS_diretory_attr which in turn is again part of a larger structure called PVFS_object_attr. PVFS_object_attr is used to store the attributes for all available objects. These structures are shown in listing 4.6. The PVFS_object_attr structure stores common metadata like owner, group, perms, atime, mtime and ctime that can be set on any object. objtype is used to differentiate between object types, which can be any of PVFS_TYPE_NONE, PVFS_TYPE_METAFILE, PVFS_TYPE_DATAFILE, PVFS_TYPE_DIRECTORY, PVFS_TYPE_SYMLINK or PVFS_TYPE_DIRDATA as defined in include/pvfs2-types.h. The union u stores the object-specific attributes. For example, if objtype is set to PVFS_TYPE_DIRECTORY, u.dir.hint would be used to store the directory hints.

Listing 4.6: Excerpt from `src/proto/pvfs2-attr.h`

```
1  /* attributes specific to directory objects */
2  struct PVFS_directory_attr_s
3  {
4      PVFS_size dirent_count;
5      PVFS_directory_hint hint;
6  };
7  typedef struct PVFS_directory_attr_s PVFS_directory_attr;
8
9  ...
10
11 /* generic attributes; applies to all objects */
12 struct PVFS_object_attr
13 {
14     PVFS_uid owner;
15     PVFS_gid group;
16     PVFS_permissions perms;
17     PVFS_time atime;
18     PVFS_time mtime;
19     PVFS_time ctime;
20     uint32_t mask;      /* indicates which fields are currently valid
       ↪ */
21     PVFS_ds_type objtype; /* defined in pvfs2-types.h */
22     union
23     {
24         PVFS_metafile_attr meta;
25         PVFS_datafile_attr data;
26         PVFS_directory_attr dir;
27         PVFS_symlink_attr sym;
28     }
29     u;
30 };
```

To make PVFS actually transfer the new directory hint between client and server the encoding/decoding functions of the affected structure – that is, `PVFS_directory_hint` – has to be adapted. These are semi-automatically created by using the `endecode` macros defined in `src/proto/endecode-funcs.h`, as shown in listing 4.7.

Listing 4.7: Excerpt from `src/proto/pvfs2-attr.h`

```
1  #ifdef _PINT_REQPROTO_ENCODE_FUNCS_C
2  endecode_fields_9(PVFS_directory_hint,
3          uint32_t, dist_name_len,
4          skip4,,
5          string, dist_name,
6          uint32_t, dist_params_len,
7          skip4,,
```

```
8           string, dist_params,
9           uint32_t, dfile_count,
10          skip4,,
11          uint32_t, no_metafile)
12  #endif
```

The `endecode_fields_9` macro accepts a variable type – in this case `PVFS_directory_hint` – and 9 fields – one variable type and one variable per field – and in turn creates two inline functions, one to encode the 9 given fields and another one to decode them again. The resulting functions are shown in listing 4.8.

Listing 4.8: Automatically generated encode/decode functions

```
1   static inline void
2   encode_PVFS_directory_hint
3       (char **pptr, const PVFS_directory_hint *x)
4   {
5       encode_uint32_t(pptr, &x->dist_name_len);
6       encode_skip4(pptr, &x->);
7       encode_string(pptr, &x->dist_name);
8       encode_uint32_t(pptr, &x->dist_params_len);
9       encode_skip4(pptr, &x->);
10      encode_string(pptr, &x->dist_params);
11      encode_uint32_t(pptr, &x->dfile_count);
12      encode_skip4(pptr, &x->);
13      encode_uint32_t(pptr, &x->no_metafile);
14  }
15
16  static inline void
17  decode_PVFS_directory_hint
18      (char **pptr, PVFS_directory_hint *x)
19  {
20      decode_uint32_t(pptr, &x->dist_name_len);
21      decode_skip4(pptr, &x->);
22      decode_string(pptr, &x->dist_name);
23      decode_uint32_t(pptr, &x->dist_params_len);
24      decode_skip4(pptr, &x->);
25      decode_string(pptr, &x->dist_params);
26      decode_uint32_t(pptr, &x->dfile_count);
27      decode_skip4(pptr, &x->);
28      decode_uint32_t(pptr, &x->no_metafile);
29  }
```

`encode_PVFS_directory_hint` from listing 4.8 takes a filled `PVFS_directory_hint` structure and encodes the contained information into a string, that is, a `char` array. `decode_PVFS_directory_hint` does the opposite: it reads a string and fills a given

`PVFS_directory_hint` structure with the decoded information. It should be noted that the other encode/decode functions used for `uint32_t`, `skip4` and `string` are predefined in `src/proto/endecode-funcs.h`.

Additionally, the new directory hint must be considered when copying objects' attributes. To do this, the function `PINT_copy_object_attr` must be changed, as shown in listing 4.9. This functions decides which fields of the structure should be copied by looking at the `mask` field of the `src` structure. Both `src` and `dest` are `PVFS_object_attr` structures. If the `PVFS_ATTR_DIR_HINT` bit is set in `mask` all directory hints are copied from `src` to `dest`.

Listing 4.9: Excerpt from `src/common/misc/pint-util.c`

```
1 if (src->mask & PVFS_ATTR_DIR_HINT)
2 {
3     dest->u.dir.hint.dfile_count =
4         src->u.dir.hint.dfile_count;
5     dest->u.dir.hint.no_metafile =
6         src->u.dir.hint.no_metafile;
7
8     ...
9 }
```

4.2.3 Server Modifications

The `get-attr` server state machine must also be modified to send the `no_metafile` directory hint back to the client. First it has to be added to the list of directory hints in `src/server/pvfs2-server.h`, as shown in listing 4.10. Also, the total number of directory hints has to be corrected by incrementing `NUM_SPECIAL_KEYS`.

Listing 4.10: Excerpt from `src/server/pvfs2-server.h`

```
1  /* optional; user-settable keys */
2  enum
3  {
4      DIST_NAME_KEY          = 0,
5      DIST_PARAMS_KEY        = 1,
6      NUM_DFILES_KEY         = 2,
7      NO_METAFILE_KEY        = 3,
8      NUM_SPECIAL_KEYS       = 4, /* not an index */
9      METAFILE_HINT_KEY      = 4,
10 };
```

Listing 4.11 and listing 4.12 show the changes needed in the `get-attr` server state machine. The directory hint's name as an extended attribute and length have to be added to the `Trove_Special_Keys` array, as shown in listing 4.11. These changes cause the directory hint to be read by Trove.

Listing 4.11: Excerpt from `src/server/get-attr.sm`

```
1  PINT_server_trove_keys_s Trove_Special_Keys[] =
2  {
3      {"user.pvfs2.dist_name"    , SPECIAL_DIST_NAME_KEYLEN},
4      {"user.pvfs2.dist_params", SPECIAL_DIST_PARAMS_KEYLEN},
5      {"user.pvfs2.num_dfiles"   , SPECIAL_NUM_DFILES_KEYLEN},
6      {"user.pvfs2.no_metafile", SPECIAL_NO_METAFILE_KEYLEN},
7      {"user.pvfs2.meta_hint"    , SPECIAL_METAFILE_HINT_KEYLEN},
8  };
```

Listing 4.12 shows a shortened version of the conversion process needed because the directory hint's value is returned as a string – that is, a `char` array. It must be converted to an integer with the `strtol` function before it gets sent back to the client. It can also be seen that the directory hint is set to `0` if it could not be read or an error occurs while converting. Therefore any value not equal to `0` can be interpreted as a set directory hint. `s_op->resp.getattr.attr` contains the `PVFS_object_attr` structure that gets sent back to the client.

Listing 4.12: Excerpt from `src/server/get-attr.sm`

```
1  long int no_metafile = 0;
2
3  if (s_op->u.getattr.err_array[NO_METAFILE_KEY] == 0)
4  {
5      char *endptr = NULL;
6      ...
7      no_metafile = strtol(s_op->val_a[NO_METAFILE_KEY].buffer,
           ↪ &endptr, 10);
8      if (*endptr != '\0' || no_metafile < 0)
9      {
10         no_metafile = 0;
11     }
12     ...
13 }
14
15 s_op->resp.u.getattr.attr.u.dir.hint.no_metafile = no_metafile;
```

4.2.4 Client Modifications

To avoid the need of having to set the `no_metafile` directory hint on each created directory the hint is inherited from the parent directory, that is, the hint is set on a newly created directory if and only if it is already set on its parent directory. This is done in the `mkdir` client state machine, as shown in listing 4.13. `sm_p->getattr.attr` contains the attributes of the parent directory. As can be seen, if the `no_metafile` directory hint is set to any value but `0` the directory hint is also set on the newly created directory. A value of `0` means that the directory hint is not set at all. This is done to make it easier to create whole directory hierarchies with directory hints. All other directory hints are inherited, too.

Listing 4.13: Excerpt from `src/client/sysint/sys-mkdir.sm`

```
1 if(sm_p->getattr.attr.u.dir.hint.no_metafile != 0)
2 {
3     gossip_debug(GOSSIP_CLIENT_DEBUG, "mkdir: setting
      ↪ no_metafile\n");
4
5     ...
6 }
```

4.2.5 Hint Usage Instructions

The now introduced directory hint can now, for example, be set with the `pvfs2-xattr` command. To set it on the directory `/pvfs2/testdir` the full command looks as follows:

```
$ pvfs2-xattr -s -k user.pvfs2.no_metafile -v 1 /pvfs2/testdir
```

It can be unset by providing a value of `0`:

```
$ pvfs2-xattr -s -k user.pvfs2.no_metafile -v 0 /pvfs2/testdir
```

It is also possible to mount a PVFS volume as a normal file system via a kernel module. Then the normal command line tools for extended attributes can be used to set the hints, too.[4] However, this is out of the scope of this thesis.

4.3 Compatibility with Unrelated State Machines

To work with the changes described in the previous sections several changes to seemingly unrelated state machines must be made. The adaptions necessary for each state machine are explained in this section.

4.3.1 Path Lookup: `lookup` State Machines

The `lookup` client and server state machines resolve a path incrementally. Therefore, a given path is split up into so-called path segments. For example, the path `/testdir/testfile` would be processed in the following way:

1. The root's (`/`) attributes are fetched.
   It is determined that this is a directory, so the directory entries are read and searched for `testdir`.
2. The directory's (`/testdir`) attributes are fetched
   It is determined that this is a directory, so the directory entries are read and searched for `testfile`.
3. The file's (`/testdir/testfile`) attributes are fetched.
   It is determined that this is a file, so the lookup ends here.

[4] See `man getfattr` and `man setfattr`.

However, the `lookup` state machines expects the last segment – that is, `testfile` – to be a metafile object and aborts otherwise. This must be prevented, therefore several changes are needed. These are shown in listing 4.14 and listing 4.15. In both cases a new case for datafile objects is introduced. This is done by an appropriate handling of the object type `PVFS_TYPE_DATAFILE`. Without this change, if a datafile object is encountered during path lookup the client aborts without sending a request to the server and the server itself simply crashes because of a failed `assert` statement.

Listing 4.14 shows the change for the `lookup` client state machine. As can be seen, datafile objects are just handled like metafile objects. This is possible, because nothing special is done if a metafile object is encountered, but rather just some generic tests. The new handling for datafile objects in the `lookup` server state machine is shown in listing 4.15. Again, datafile objects are handled just like metafile objects, in fact the code is a slightly modified version of the metafile object handling code.

Listing 4.14: Excerpt from `src/client/sysint/sys-lookup.sm`

```
1  switch(cur_seg->seg_attr.objtype)
2  {
3      case PVFS_TYPE_DIRECTORY:
4          js_p->error_code = LOOKUP_TYPE_DIRECTORY;
5          break;
6      case PVFS_TYPE_DATAFILE:
7      case PVFS_TYPE_METAFILE:
8          js_p->error_code = LOOKUP_TYPE_METAFILE;
9          break;
10
11     ...
```

Listing 4.15: Excerpt from `src/server/lookup.sm`

```
1  if (a_p->objtype == PVFS_TYPE_DATAFILE)
2  {
3      gossip_debug(GOSSIP_SERVER_DEBUG, "  object is a datafile; "
4                   "halting lookup and sending response\n");
5
6      js_p->error_code = STATE_ENOTDIR;
7      return 1;
8  }
```

4.4 File Creation: `create` Client State Machine

In its original form, the `create` client state machine does the following to create a new file in a given directory:

1. Get the parent directory's attributes

- Amongst other things, get the directory hints

2. Create the metafile object
3. Create the datafile objects
4. Set the metafile object's attributes
   - Set the datafile objects' handles
5. Create a directory entry for the new file
   - Insert the metafile object's handle into the directory data object

Since the parent directory's hints are read in the first step, they can be used to decide the further action based on whether the `no_metafile` directory hint is set or not. To force the creation of only one datafile object the existing code to request an arbitrary number of datafile objects is modified, as shown in listing 4.16. Independent of the requested number of datafile objects the limit is set to `1`, thus ensuring that the file system remains in a consistent state. This does not limit the file size, but only the striping factor. It must be noted that all file data ends up on a single data server and therefore this probably is only feasible for small files.

Listing 4.16: Excerpt from `src/client/sysint/sys-create.sm`

```
1 if (attr->u.dir.hint.no_metafile != 0)
2 {
3     /*
4      * Always create only one datafile.
5      */
6     sm_p->u.create.num_data_files = 1;
7 }
```

To decide whether or not to skip a state the attributes of the parent directory are checked, as shown in listing 4.17. New transitions are added to the state machine, such that the newly introduced error code of `CREATE_SHORTCUT` causes the state machine to skip this particular state. This is used to skip the states that create the metafile object and set the metafile object's attributes.

Listing 4.17: Excerpt from `src/client/sysint/sys-create.sm`

```
1 if (sm_p->getattr.attr.u.dir.hint.no_metafile != 0)
2 {
3     /*
4      * Skip this state.
5      */
6     js_p->error_code = CREATE_SHORTCUT;
7     return 1;
8 }
```

To keep the changes as small as possible the existing code that inserts the metafile object's handle as a new directory entry is reused. However, the metafile object's handle is simply set to the datafile object's handle such that it gets used instead. This is shown in listing 4.18.

Listing 4.18: Excerpt from `src/client/sysint/sys-create.sm`

```
1 if (sm_p->getattr.attr.u.dir.hint.no_metafile != 0)
2 {
3     /*
4      * We set metafile_handle so the datafile_handle gets inserted
           ↪ into
5      * the directory.
6      */
7     sm_p->u.create.metafile_handle =
          ↪ sm_p->u.create.datafile_handles[0];
8 }
```

If the hint is set, all steps involving the metafile object are skipped, that is, steps two and four. The resulting process looks like this:

1. Get the parent directory's attributes
   - Amongst other things, get the directory hints
2. Create the datafile object
3. Create a directory entry for the new file
   - Insert the datafile object's handle into the directory

As can be seen, two of of a total of five steps are skipped. Instead of the metafile object's handle the datafile object's handle can be used, since it is ensured that only one datafile object is created.

4.5 File Listing: `getattr` Client State Machine

The `getattr` client state machine is a nested state machine that requests the attributes of a given object from the server and returns them to the calling state machine. Therefore it has the potential to make many operations "just work" even with the completely different situation of a missing metafile object. It is modified to fake the attributes of a metafile object in case the attributes of a file created with the `no_metafile` directory hint is requested. The only information known about the datafile object on the server side is its handle and its size, all other information must therefore be faked. This information includes:

- Common metadata
  - Ownership
  - Permissions
  - Access, modification and change time

- Metafile object-specific attributes
  - Distribution function and parameters
  - The datafile objects' handles
  - Number of datafile objects

With some additional effort it would also be possible to selectively set some metadata on the datafile object and fake the remaining metadata. To do this, the `setattr` call in the `create` client state machine must be adapted to set only the desired metadata. However, as already mentioned, this is not done here for performance reasons.

The `getattr` client state machine takes different actions based on the object type. If a datafile object's attributes are returned it simply does nothing. However, the code in listing 4.19 causes it to fake the attributes of a metafile object if required. The attributes may only be faked if a metafile object's attributes are actually requested, because it is perfectly legal to request a datafile object's attributes to determine the datafile object's size. `sm_p->getattr.req_attrmask` stores which attributes are requested. If all possible attributes of a metafile object are requested with `PVFS_ATTR_META_ALL`, it can be assumed that a metafile object's attributes are requested. Therefore the attributes may then be faked. The common metadata is faked in such a way that the file appears to belong to the user and group with ID `0`, that is, `root`. The permissions are set to `-rw-rw-rw-`[5], which means that anyone may read or write the file. The different times are all set to `0`, that is, the beginning of the Unix epoch.[6] It is important to mention that this is only used for display purposes and the like and is not the actual metadata set on the datafile object.

Listing 4.19: Excerpt from `src/client/sysint/sys-getattr.sm`

```
1  switch (attr->objtype)
2  {
3      ...
4
5      case PVFS_TYPE_DATAFILE:
6          /*
7           * If a metafile was requested, this is probably a file
                ↪ created
8           * with no_metafile.
9           */
10         if (sm_p->getattr.req_attrmask & PVFS_ATTR_META_ALL)
11         {
12             /*
13              * Manually fill out all details for a metafile.
14              */
15             PINT_SM_DATAFILE_SIZE_ARRAY_INIT(
                    ↪ &sm_p->getattr.size_array, 1);
16             sm_p->getattr.size_array[0] = attr->u.data.size;
17
```

[5] See `man chmod` for detailed information.
[6] 1970-01-01 00:00:00Z

```
18              attr->owner = 0;
19              attr->group = 0;
20              attr->perms = 0666;
21              attr->atime = 0;
22              attr->mtime = 0;
23              attr->ctime = 0;
24
25              attr->mask |= PVFS_ATTR_COMMON_ALL | PVFS_ATTR_META_ALL
                    ↪ | PVFS_ATTR_DATA_ALL;
26              attr->objtype = PVFS_TYPE_METAFILE;
27
28              attr->u.meta.dfile_count = 1;
29              attr->u.meta.dfile_array = malloc(sizeof(PVFS_handle));
30              attr->u.meta.dfile_array =
                    ↪ memset(attr->u.meta.dfile_array, 0,
                    ↪ sizeof(PVFS_handle));
31              attr->u.meta.dfile_array[0] =
                    ↪ sm_p->getattr.object_ref.handle;
32              attr->u.meta.dist =
                    ↪ PINT_dist_create(PVFS_DIST_BASIC_NAME);
33              attr->u.meta.dist_size = PVFS_DIST_BASIC_NAME_SIZE;
34          }
35          return 0;
36
37      ...
```

4.6 File Removal: `remove` Client State Machine

In its original form, the client `remove` state machine does the following to remove a file from a given directory:

1. Remove the file's directory entry
   - Remove the metafile object's handle from the directory
2. Get the metafile object's attributes
   - The datafile objects' handles are needed
3. Remove the datafile objects
4. Remove the metafile object

Because of the changes to the `getattr` client state machine, attributes of a metafile object are returned even if the file was created with the `no_metafile` directory hint set. However, if the returned metafile object's handle equals the handle of its only datafile object, the removal of the metafile object can be skipped, as shown in listing 4.20. Because these handles are globally unique it is ensured that they indeed refer to the same object. A new transition is added to

the state machine, such that the newly introduced error code of `REMOVE_SHORTCUT` causes the state machine to skip the metafile object's removal.

Listing 4.20: Excerpt from `src/client/sysint/remove.sm`

```
1 if (sm_p->object_ref.handle ==
    ↪ sm_p->getattr.attr.u.meta.dfile_array[0])
2 {
3     /*
4      * The metafile's handle equals the datafile's handle.
5      * Skip this state.
6      */
7     js_p->error_code = REMOVE_SHORTCUT;
8     return 1;
9 }
```

After these changes, the resulting process looks like this:

1. Remove the file's directory entry
   - Remove the datafile object's handle from the directory
2. Get the datafile object's attributes
   - Returns metafile object's faked attributes
3. Remove the datafile object

As can be seen, one of a total of four steps is skipped. Step two can not be skipped, because it is not known beforehand whether the file was created with `no_metafile` or not.

Summary

The changes introduced due to the implementation of the metadata optimizations are kept minimal. Existing code and infrastructure is re-used as much as possible to ease future development and reduce the chance of incompatibilities. All in all, the code changes are relatively straightforward. However, the initial familiarization with the code is more challenging, because it involves reading a lot of debug output and comments in the code are sometimes lacking.

5 Benchmarking

To measure the benefits of the changes described in the previous chapters a relatively simple benchmark program is designed. The program creates, lists and removes a big amount of files in a relatively flat directory hierarchy. For details on this program, see appendix B.

To simulate several different environments the number of concurrently accessing clients and the underlying storage are varied. Moderate load is simulated by only one client accessing the file system, while five concurrent clients simulate heavy load. The clients are independent instances of the benchmark program. To observe the influence of disk latency, PVFS's storage space is put into a normal directory on an `ext3` partition and in RAM, that is, its own `tmpfs` partition. This is especially important as PVFS – in its default configuration – forces metadata modifications to disk.[1] Consequently, disk latency plays an important role in the overall performance.

5.1 Hardware Configurations

To test if the metadata optimizations work well in different environments, two basic configurations are tested. One configuration uses only a single machine, therefore eliminating any influence of the network. The other configuration uses multiple machines, thus network latency as well as disk latency influence the performance.

5.1.1 Single Machine

The clients, data and metadata server all run on the same machine with an Intel Core 2 Duo 2.4 GHz, 2 GByte RAM and a SATA disk.

5.1.2 Five Machines

Five machines from the cluster of the workgroup "Parallel and Distributed Systems" are used. Two machines act as data servers, another two as metadata servers and a fifth machine is used for the clients. Each machine is equipped with two Intel Xeon 2.0 GHz, 1 GByte RAM, an ATA disk and a 1 GBit/s network interface.

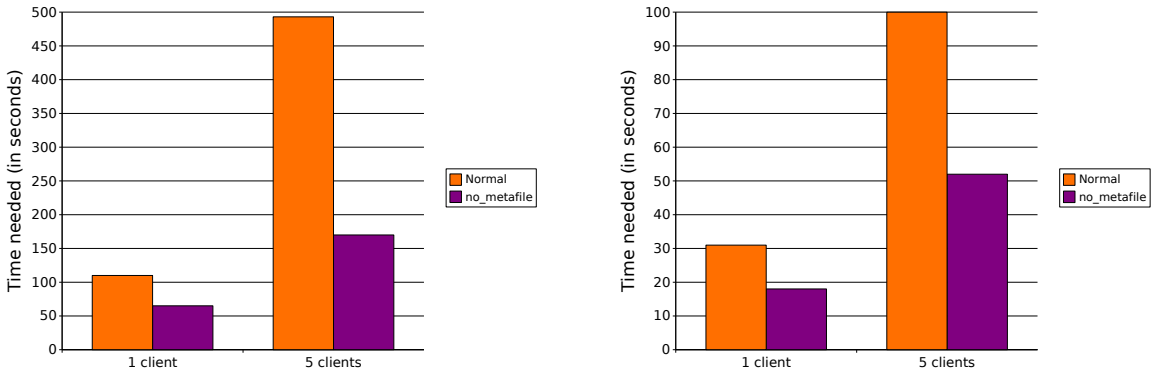
[1] For more information, see `man sync` and `man fsync`.

5.2 Single Machine

5.2.1 File Creation

The benchmark program creates 100 child directories in a single parent directory and populates each with 500 files. Only the time needed to create these 50,000 files is measured, the directories are created before the actual benchmark starts. To exclude the influence of the `io` client state machine, files of size 0 are created.

Figure 5.1a shows the time each client needs to create 50,000 files, once with and once without the `no_metafile` directory hint set. PVFS's storage space is put in a normal directory on an `ext3` partition. In figure 5.1b the same values as in the last one are shown, except that PVFS's storage space is put on its own `tmpfs` partition, thus removing any latencies the disk introduces.



(a) On disk

(b) On `tmpfs`

Figure 5.1: File creation

As can be seen in figure 5.1a, if only one client writes to the file system and the `no_metafile` directory hint is set, the time needed to create the 50,000 files decreases to about 60% of the time needed to create them without the hint. However, if five clients work concurrently the time decreases to about 35% of the original. This is probably due to the fact that metadata writes are by default synchronous. However, these are exactly the operations that are skipped if `no_metafile` is set and thus the server can process more requests in parallel instead of waiting for the slow disk.

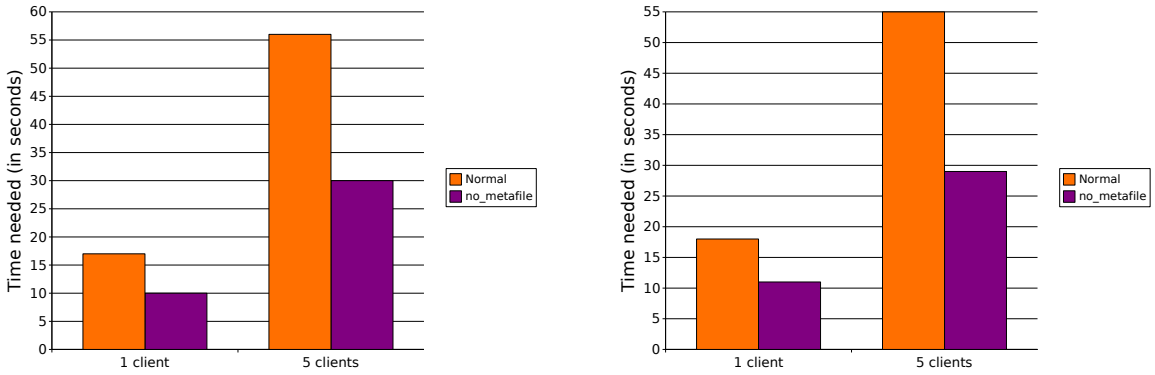
In figure 5.1b the speedup with five concurrent clients is less drastic, because no disk seek times could be avoided.

5.2.2 File Listing

The program lists the files in each directory such that details like permissions, ownership etc. are shown, too. This is done so that the client has to contact each data file's server, because

otherwise only the names would need to be fetched from the metadata server. In particular the `-l` flag of `pvfs2-ls` is used.[2]

Figure 5.2a shows the time each client needs to list the 50,000 files, once with and once without the `no_metafile` directory hint set. PVFS's storage space is put in a normal directory on an `ext3` partition. In figure 5.2b the same values as in the last one are shown, except that PVFS's storage space is put on its own `tmpfs` partition, thus removing any latencies the disk introduces.



(a) On disk

(b) On `tmpfs`

Figure 5.2: File listing

As can be seen in 5.2a, if only one client reads from the file system and the `no_metafile` directory hint is set, the time needed to list the 50,000 files decreases to about 60% of the time needed to list them without the hint. With five concurrent clients the time decreases to about 55% of the original.

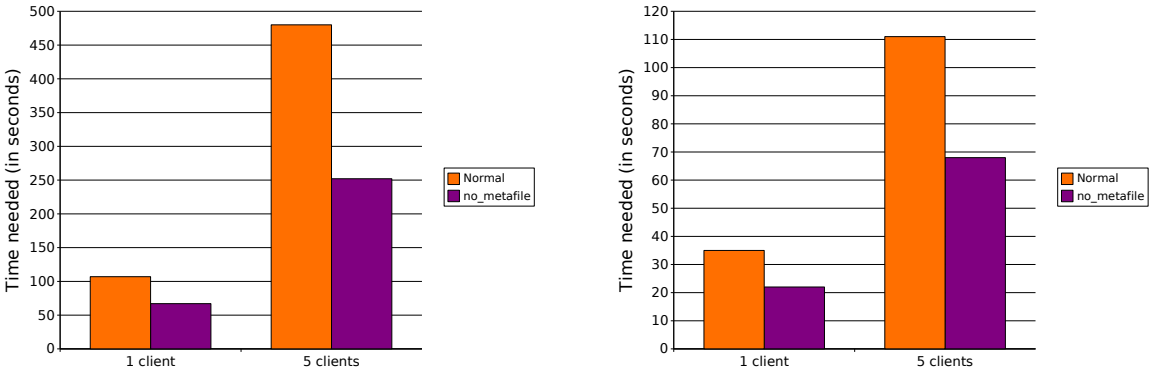
In figure 5.2b for some reason the single client takes longer than in the previous case. The time for five concurrent clients is nearly identical, however.

5.2.3 File Removal

The program removes all files and directories such that the file system is in the same state as before the benchmark was started. Only the time needed to remove the 50,000 files is measured, the directories are removed after the actual benchmark ends.

Figure 5.3a shows the time each client needs to remove the 50,000 files, once with and once without the `no_metafile` directory hint set. PVFS's storage space is put in a normal directory on an `ext3` partition. In figure 5.3b the same values as in the last one are shown, except that PVFS's storage space is put on its own `tmpfs` partition, thus removing any latencies the disk introduced.

[2] Compare the output of `ls` and `ls -l` on any Unix system to see the difference.



(a) On disk

(b) On `tmpfs`

Figure 5.3: File removal

As can be seen in figure 5.3a, if only one client at a time is running and the `no_metafile` directory hint is set, the time needed to remove the 50,000 files decreases to about 65% of the time needed to remove them without the hint. However, if five clients run concurrently the time decreases to about 50% of the original. This is probably due to the fact that metadata writes are by default synchronous. However, these are exactly the operations that are skipped if `no_metafile` is set and thus the server can process more requests in parallel instead of waiting for the slow disk.

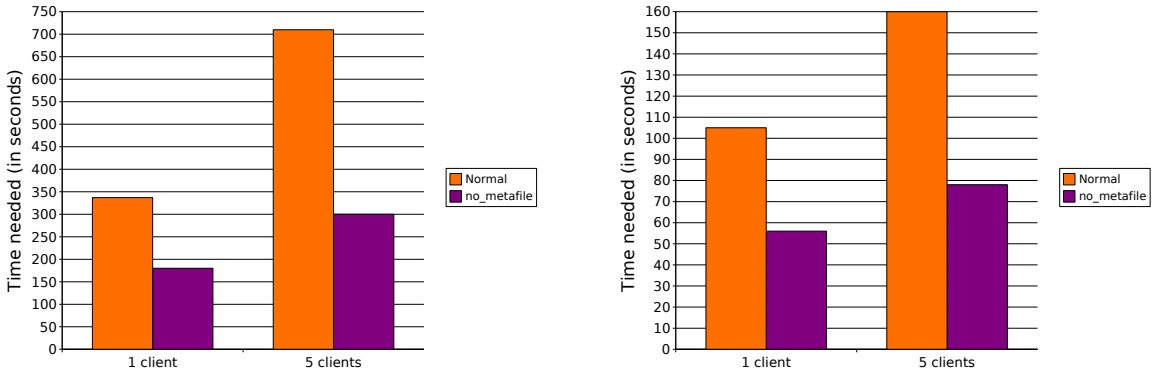
In figure 5.3b for the first time there is no speedup to be gained by increasing the client concurrency, but rather a slight slowdown.

5.3 Five Machines

5.3.1 File Creation

The program works as described in subsection 5.2.1.

Figure 5.4a shows the time each client needs to create 50,000 files, once with and once without the `no_metafile` directory hint set. PVFS's storage space is put in a normal directory on an `ext3` partition. In figure 5.4b the same values as in the last one are shown, except that PVFS's storage space is put on its own `tmpfs` partition, thus removing any latencies the disk introduces.



(a) On disk

(b) On `tmpfs`

Figure 5.4: File creation

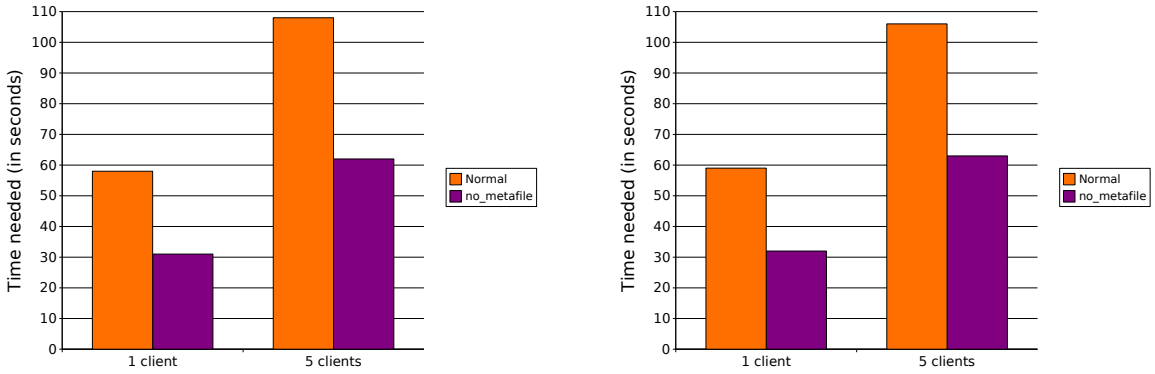
As can be seen in figure 5.4a, if only one client writes to the file system and the `no_metafile` directory hint is set, the time needed to create the 50,000 files decreases to about 50% of the time needed to create them without the hint. This is even better than with only one machine, where the network is not used at all. However, if five clients work concurrently the time decreases to about 40% of the original. This is slightly slower than with one machine and therefore there is less speedup to be gained by increasing client concurrency. The speedup increase is probably due to the fact that metadata writes are by default synchronous. However, these are exactly the operations that are skipped if `no_metafile` is set and thus the server can process more requests in parallel instead of waiting for the slow disk. Since this time the network is used, it can be seen that disk latency still plays an important role in terms of performance, even with the additional network latency.

In figure 5.4b the speedup with five concurrent clients is less drastic, because no disk seek times could be avoided.

5.3.2 File Listing

The program works as described in subsection 5.2.1.

Figure 5.5a shows the time each client needs to list the 50,000 files, once with and once without the `no_metafile` directory hint set. PVFS's storage space is put in a normal directory on an `ext3` partition. In figure 5.5b the same values as in the last one are shown, except that PVFS's storage space is put on its own `tmpfs` partition, thus removing any latencies the disk introduces.



(a) On disk

(b) On `tmpfs`

Figure 5.5: File listing

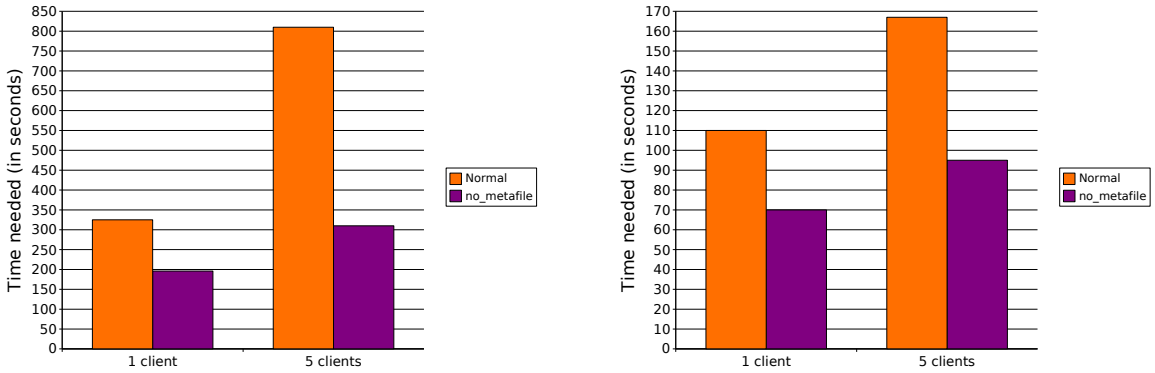
As can be seen in figure 5.5a, if only one client reads from the file system and the `no_metafile` directory hint is set, the time needed to list the 50,000 files decreases to about 50% of the time needed to list them without the hint. With five concurrent clients the time increases to about 55% of the original. This is one of the rare cases where an increase in client concurrency does not improve the speedup. Since only metadata reads are needed for this file system operation and therefore no slow metadata writes could be skipped, there are no huge performance gains possible by reducing the impact of disk latency. In contrast to metadata writes, these metadata reads can be sped up by using the file system cache. The optimized version only does one metadata read instead of two metadata reads and since they usually are fast because of caching, network latency outweighs the benefits of the one skipped metadata read.

In figure 5.5b the times are nearly identical to the ones in figure 5.5a.

5.3.3 File Removal

The program works as described in subsection 5.2.3.

Figure 5.6a shows the time each client needs to remove the 50,000 files, once with and once without the `no_metafile` directory hint set. PVFS's storage space is put in a normal directory on an `ext3` partition. In figure 5.6b the same values as in the last one are shown, except that PVFS's storage space is put on its own `tmpfs` partition, thus removing any latencies the disk introduced.



(a) On disk

(b) On `tmpfs`

Figure 5.6: File removal

As can be seen in figure 5.6a, if only one client at a time is running and the `no_metafile` directory hint is set, the time needed to remove the 50,000 files decreases to about 60% of the time needed to remove them without the hint. This is even better than with only one machine, where the network is not used at all. However, if five clients run concurrently the time decreases to about 40% of the original. This is significantly faster than with one machine and therefore there is more speedup to be gained by increasing client concurrency. The speedup increase is probably due to the fact that metadata writes are by default synchronous. However, these are exactly the operations that are skipped if `no_metafile` is set and thus the server can process more requests in parallel instead of waiting for the slow disk. Since this time the network is used, it can be seen that disk latency still plays an important role in terms of performance, even with the additional network latency.

In figure 5.6b the speedup with five concurrent clients is less drastic, because no disk seek times could be avoided.

5.4 Summary

Figure 5.7 shows an overview of the efficiency of the metadata optimizations as measured in section 5.2 with a single machine. For each operation – that is, creation, listing and removal – the percentage of time needed for completion with the `no_metafile` hint set in comparison to the time needed without it is shown. Also, for each operation the efficiencies for a varying number of concurrent clients and underlying file systems are shown in detail.

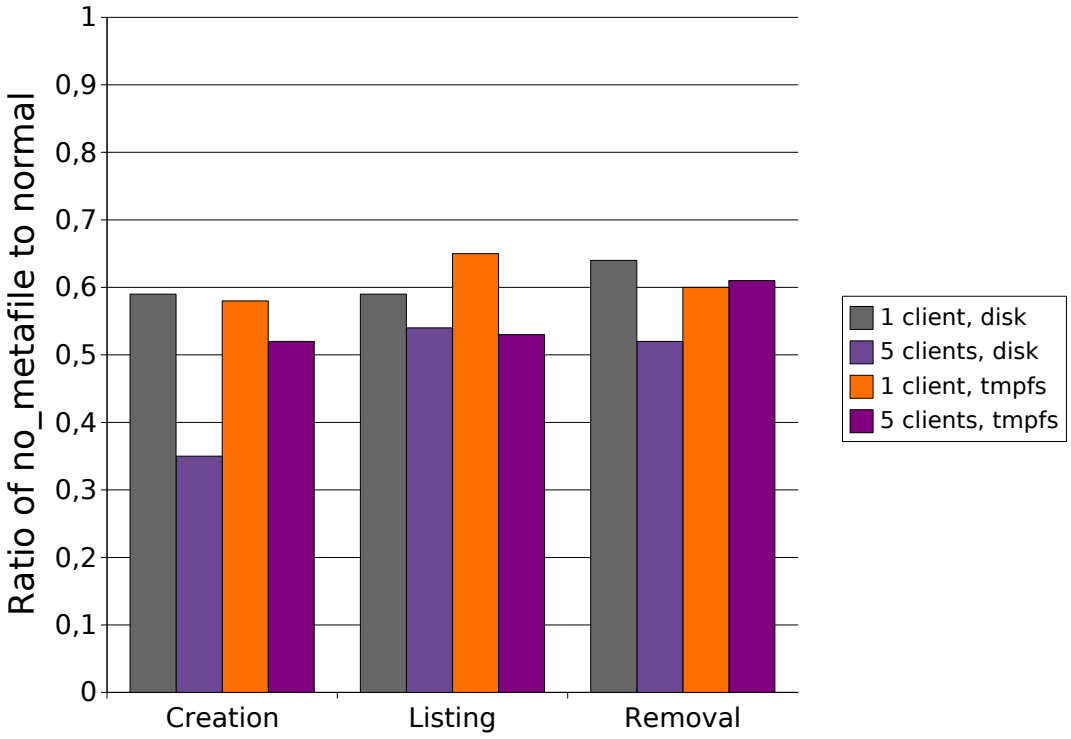


Figure 5.7: Efficiency of the optimized file system operations

As can be seen in figure 5.7, almost all operations benefit from an increase in concurrency. This is especially true for the creation operation where each client is almost twice as fast if five of them are running at the same time. The benefit is less pronounced on `tmpfs` partitions, since in this case there are no slow synchronous disk operations that could be skipped. The listing operation also does not benefit from any disk-related savings, but only from one less message pair per file, so the speedup here is lower. As shown in chapter 3 and chapter 4 about half of the work in each of the three file system operations is skipped, therefore the performance gains are within expected boundaries or – as is the case with file creation – even surpass the expectations.

Figure 5.8 shows an overview of the efficiency of the metadata optimizations as measured in section 5.3 with five machines. The values are shown in the same way as in figure 5.7.

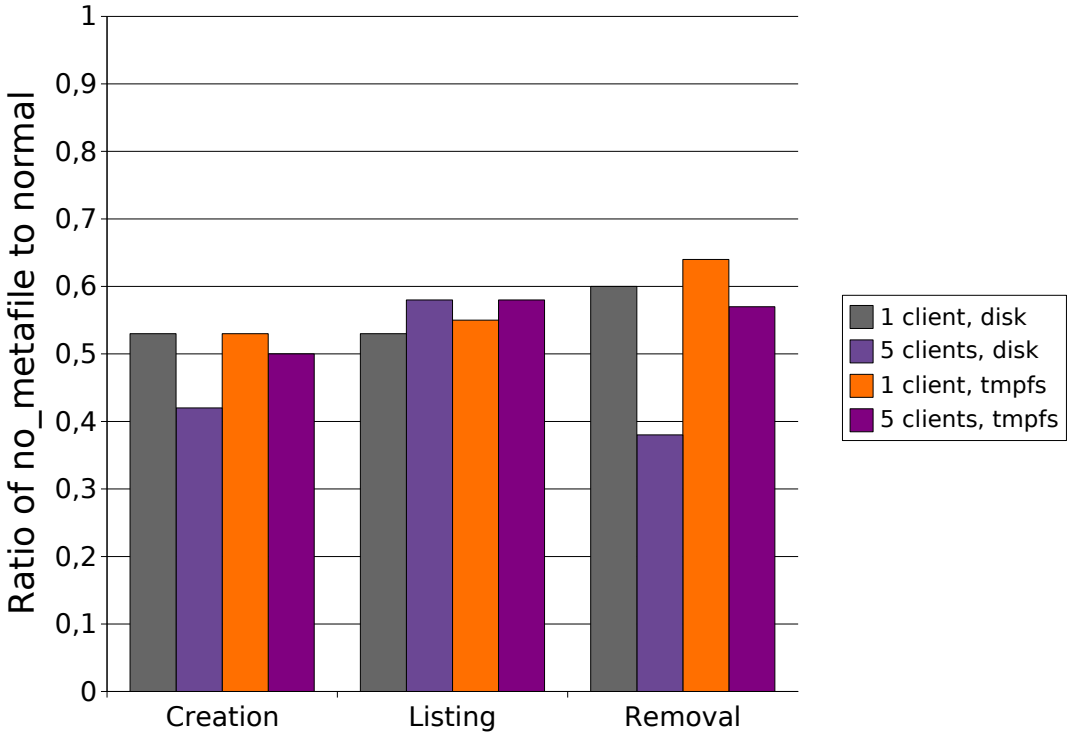


Figure 5.8: Efficiency of the optimized file system operations

As can be seen in figure 5.8, this time only disk-bound operations benefit from an increase in the concurrency, because of network effects that are not present in the local benchmarks. As in the local case, the benefit is less pronounced on `tmpfs` partitions, since in this case there are no slow synchronous disk operations that could be skipped. As shown in chapter 3 and chapter 4 about half of the work in each of the three file system operations is skipped, therefore the performance gains are within expected boundaries or – as is the case with file removal – even surpass the expectations.

Figure 5.7 and figure 5.8 show that the metadata optimizations reduce the time needed for any of the affected operations – that is, file creation, listing and removal – to about 50–60%, independent of the underlying file system. These improvements are also independent of the used hardware configuration and software environment. Especially, the speedup is approximately the same regardless of the use of the network. The disk-bound operations – that is, file creation and removal – especially benefit from these optimizations. If the underlying file system is on disk – which should be the normal case – an increase in client concurrency even reduces the time needed to 30–40%. Therefore the optimizations are especially useful for parallel access from multiple clients.

To measure the impact of the metadata optimizations from the server's point of view, the low-level operations of the server are visualized in chapter 6. This is done using a toolkit for parallel I/O visualization.

6 Visualization

A tracing facility within the PVFS server can be used to log internal operations like network communication or disk access. This is used in this chapter to visualize the impact of the metadata optimizations on the server.

To do this, an as of yet unreleased version of PIOViz[1] is used. PIOViz is a compilation of MPICH2, PVFS, several patches and a build system to automatically create an environment suitable for parallel I/O visualization. The version of PVFS shipped with PIOViz is updated to include the metadata optimizations.

Client operations can be traced, too, but at the moment the tracing is only implemented for programs that are using ROMIO to communicate with the PVFS server. For details on this, see [Kre06]. Since the PVFS client library is used directly for this visualization, no client traces are available.

6.1 Dataspace and Key-Value Pairs

In the following sections the low-level actions of the PVFS server performed for each file system operation are shown. These actions are performed by the Trove I/O layer. Trove can operate on the so-called dataspace and key-value pairs. The dataspace stores actual data of a logical file, that is, datafile objects. Key-value pairs are used to store everything else, most notably object attributes like common metadata, directory hints and so on. Functions that operate on the dataspace are prefixed by `TROVE_DSPACE`, while functions to modify key-value pairs are prefixed by `TROVE_KEYVAL`.

[1] Parallel Input/Ouput Visualization. An environment for advanced visualization of parallel I/O using MPICH2 and PVFS.

6.2 File Creation

Figure 6.1 shows the operations needed on the server to create one file in a directory without the `no_metafile` directory hint set.

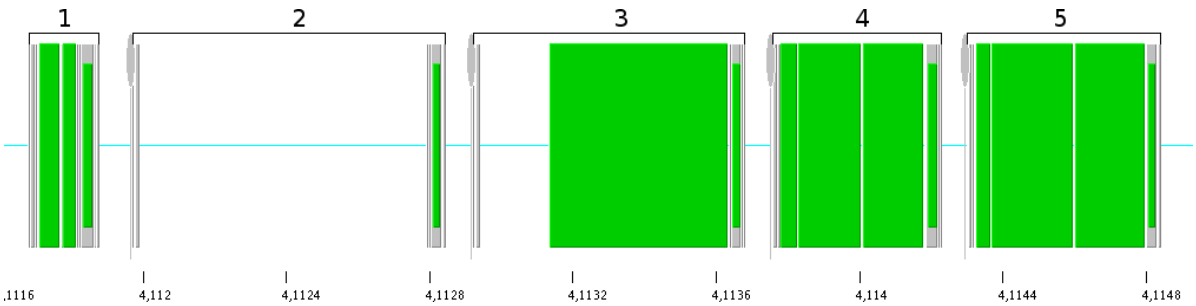


Figure 6.1: File creation without `no_metafile`

The following are the steps needed:

1. Read the parent directory's attributes
   - `TROVE_DSPACE_GETATTR`: Read attributes
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_READ_LIST`: Read directory hints
2. Create the metafile object
   - `TROVE_DSPACE_CREATE`: Create metafile object
3. Create the datafile objects
   - `TROVE_DSPACE_CREATE`: Create datafile objects
   - `TROVE_KEYVAL_WRITE`: Write parent handle
4. Write the metafile object's attributes
   - `TROVE_KEYVAL_WRITE`: Write datafile objects' handles
   - `TROVE_KEYVAL_WRITE`: Write distribution
   - `TROVE_DSPACE_SETATTR`: Write attributes
5. Create a directory entry for the new file
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_WRITE`: Write new directory entry
   - `TROVE_DSPACE_SETATTR`: Write attributes

It is obvious that steps 2 and 4 can be completely skipped, because no metafile object is created. A more detailed explanation of the steps is given in the description of figure 6.2.

Figure 6.2 shows the same just for a directory with the `no_metafile` directory hint set.

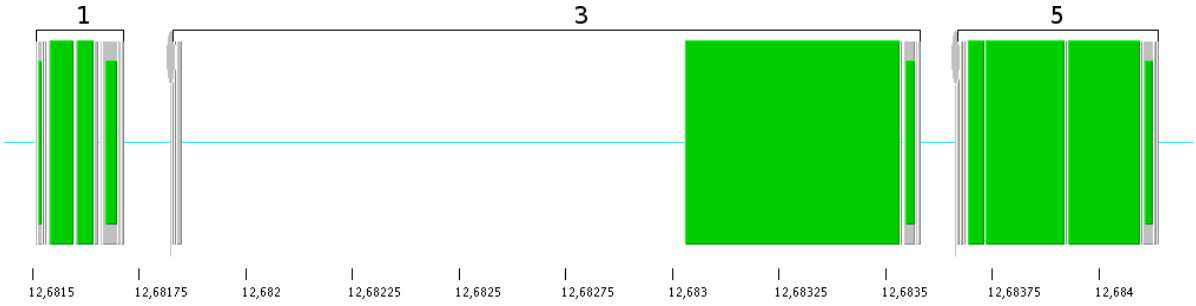


Figure 6.2: File creation with `no_metafile`

The following are the steps needed after optimization:

1. Read the parent directory's attributes
   - `TROVE_DSPACE_GETATTR`: Read attributes
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_READ_LIST`: Read directory hints
3. Create the datafile object
   - `TROVE_DSPACE_CREATE`: Create datafile object
   - `TROVE_KEYVAL_WRITE`: Write parent handle
5. Create a directory entry for the new file
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_WRITE`: Write new directory entry
   - `TROVE_DSPACE_SETATTR`: Write attributes

Note: In steps 2 and 3 the `TROVE_DSPACE_CREATE` call is not visible, because it currently does not get logged.

In step 1 it is not really necessary to read the directory data handle, because it is only needed to work with the directory entries. Since the directory entries are left untouched here, this substep could be skipped to increase performance.

In step 3 the writing of the parent handle is an addition of the load balancing code introduced in [Kun07]. Since it probably is useless in the case of no existing metafile object and also takes a considerable amount of time, it could also be skipped to increase performance.

In steps 4 and 5 no further improvements are possible.

Summary

As can be seen in figure 6.1 and figure 6.2 all actions that apply to the metafile object are skipped. Also, two additional starting points for further optimizations could be identified, namely the reading of the directory data object's handle and the writing of the datafile object's parent handle.

6.3 File Listing

Figure 6.3 shows the operations needed on the server to list a directory without the `no_metafile` directory hint set. The directory contains only one file.

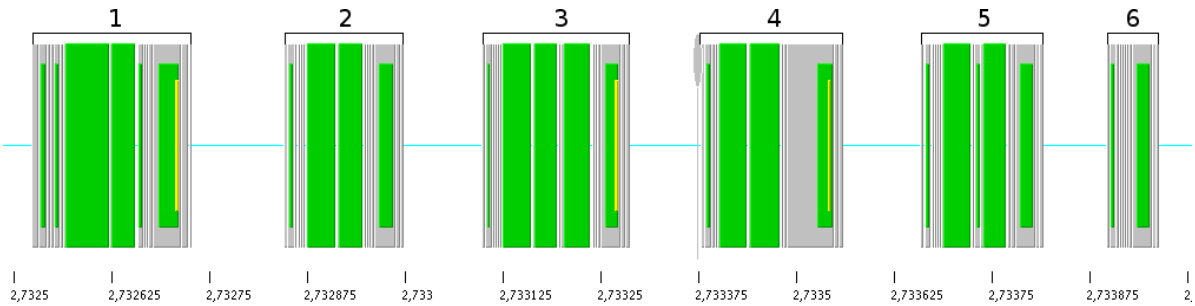


Figure 6.3: File listing without `no_metafile`

The following are the steps needed:

1. Lookup the metafile object's handle
   - `TROVE_DSPACE_GETATTR`: Read attributes
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_READ`: Read metafile object's handle
   - `TROVE_DSPACE_GETATTR`: Read attributes
2. Read the directory's attributes
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_GET_HANDLE_INFO`: Read directory entry count
3. Read the directory's attributes
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_GET_HANDLE_INFO`: Read directory entry count
   - `TROVE_KEYVAL_READ_LIST`: Read directory hints
4. List all files in the directory
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_ITERATE`: Read all files from the directory data object
5. Read the metafile object's attributes
   - `TROVE_KEYVAL_READ`: Read metafile hints
   - `TROVE_KEYVAL_READ`: Read datafile objects' handles
6. Read the datafile object's attributes

It is obvious that step 5 can be completely skipped, because no metafile object is created. A more detailed explanation of the steps is given in the description of figure 6.4.

Figure 6.4 shows the same just for a directory with the `no_metafile` directory hint set.

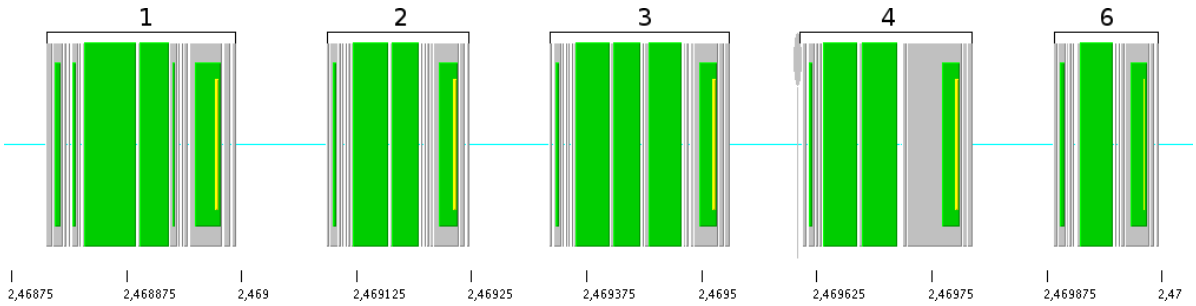


Figure 6.4: File listing with `no_metafile`

The following are the steps needed after optimization:

1. Lookup the metafile object's handle
   - `TROVE_DSPACE_GETATTR`: Read attributes
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_READ`: Read metafile object's handle
   - `TROVE_DSPACE_GETATTR`: Read attributes
2. Read the directory's attributes
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_GET_HANDLE_INFO`: Read directory entry count
3. Read the directory's attributes
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_GET_HANDLE_INFO`: Read directory entry count
   - `TROVE_KEYVAL_READ_LIST`: Read directory hints
4. List all files in the directory
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_ITERATE`: Read all files from the directory data object
6. Read the datafile object's attributes
   - `TROVE_KEYVAL_READ`: Read parent handle

Note: Steps 1 and 2 are only executed once for each directory, so their impact on performance is negligible if the directory contains more files.

In step 2 it is not really necessary to read the directory's attributes. This is done basically only to determine the object type. This step could be skipped to improve performance.

In steps 3, 4 and 5 no further optimizations are possible.

In step 6 the parent handle of the datafile object's is read when `no_metafile` is set. This is probably due to the fact that a different mask is used to read the datafile object's attributes and could be avoided to increase performance.

Summary

As can be seen in figure 6.3 and figure 6.4 the single step that applies to the metafile object is skipped. Also, two additional starting points for further optimizations could be identified, namely the redundant reading of the directory's attributes and the reading of the datafile object's parent handle.

6.4 File Removal

Figure 6.5 shows the operations needed on the server to remove one file in a directory without the `no_metafile` directory hint set.

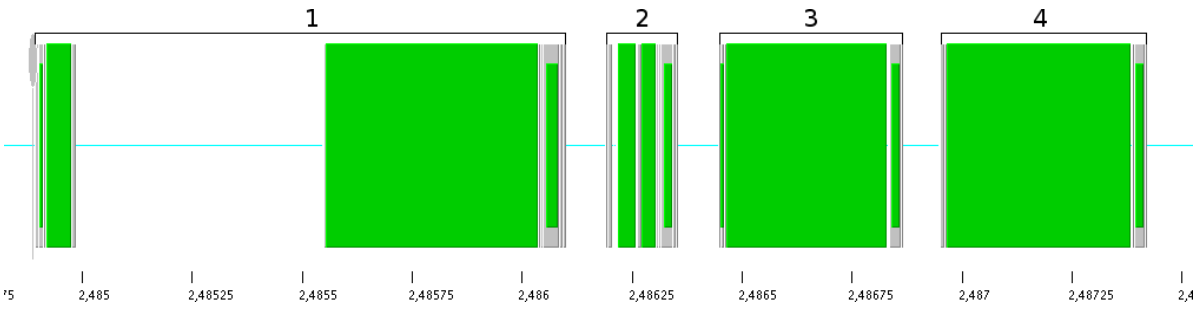


Figure 6.5: File removal without `no_metafile`

The following are the steps needed:

1. Remove the file's directory entry
   - `TROVE_KEYVAL_READ`: Read directory data object's handle
   - `TROVE_KEYVAL_REMOVE`: Remove directory entry
   - `TROVE_DSPACE_SETATTR`: Write attributes
2. Read the metafile object's attributes
   - `TROVE_KEYVAL_READ`: Read hints
   - `TROVE_KEYVAL_READ`: Read datafile objects' handles
   - `TROVE_KEYVAL_READ`: Read distribution
3. Remove the datafile objects
   - `TROVE_DSPACE_REMOVE`: Delete datafile objects
4. Remove the metafile object

– `TROVE_DSPACE_REMOVE`: Delete metafile object

It is obvious that step 4 can be completely skipped, because no metafile object must be removed. A more detailed explanation of the steps is given in the description of figure 6.6.

Figure 6.6 shows the same just for a directory with the `no_metafile` directory hint set.

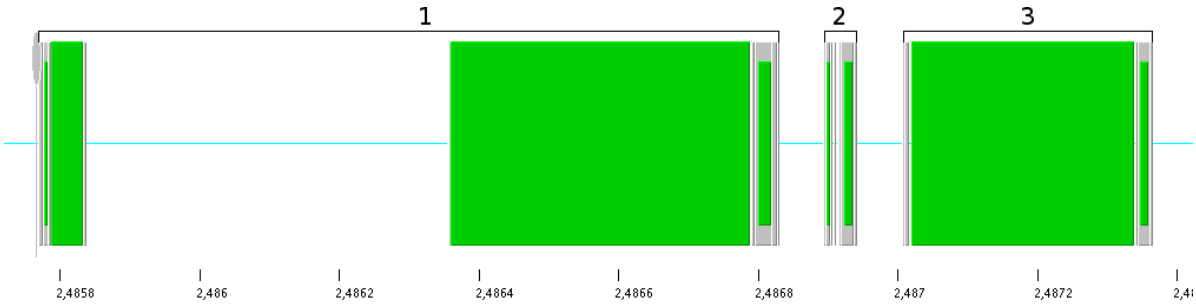


Figure 6.6: File removal with `no_metafile`

The following are the steps needed after optimization:

1. Remove the file's directory entry
   – `TROVE_KEYVAL_READ`: Read directory data object's handle
   – `TROVE_KEYVAL_REMOVE`: Remove directory entry
   – `TROVE_DSPACE_SETATTR`: Write attributes
2. Read the datafile object's attributes
3. Remove the datafile object
   – `TROVE_DSPACE_REMOVE`: Delete datafile object

Note: In step 1 the `TROVE_KEYVAL_REMOVE` call is not visible, because it currently does not get logged.

In step 2 the datafile object's size is not read, because it already is read by the so-called `prelude` server state machine that runs beforehand to check permissions and such.

In step 3 no further improvements are possible.

Summary

As can be seen in figure 6.5 and figure 6.6 the single step that applies to the metafile object is skipped. However, no additional starting points for further optimizations could be identified.

7 Summary, Conclusion and Future Work

In this chapter several possible additions to the current implementation are presented. Also, future work that could be done to improve metadata performance is evaluated.

7.1 Summary

In cluster file systems metadata operations can be very expensive. However, there are use cases where metadata is not necessary. For example, several thousand temporary files may be created, processed and deleted afterwards. To speed up metadata performance in such cases several modifications to the parallel cluster file system PVFS are made. By reducing the complexity of common metadata operations considerable speedups can be achieved.

7.2 Conclusion

While the metadata optimizations described and implemented in this thesis do not offer a speedup of several orders of magnitude, the time needed for some common file system operations could be reduced to about 50%. This achievement is quite satisfying, considering the relatively small amount of changes made. Also, since about 50% of the work in each affected file system operation is skipped, this improvements are well within expected boundaries. Together with other improvements that could be implemented in the future they could offer users of PVFS the possibility of tuning metadata operations and thus metadata performance to fit their needs.

On the other hand, these metadata optimizations change the file system semantics, because certain metadata is simply not stored. However, because they must be explicitly enabled and do not influence the normal operation of PVFS, this is not much of a concern.

7.3 Future Work

To fully integrate the changes, several other changes would need to be made. For example, the `pvfs2-fs-dump` tool currently crashes if it encounters a file created with the `no_metafile` hint set. Another example would be the error states of each affected state machine. Currently the error handling works, but could be improved to take the `no_metafile` directory hint into account. For example, if an error occurs in the `remove` client state machine the metafile object and all associated datafile objects are deleted again. Since there is no metafile object, the server does nothing. However, an additional message pair is needed to contact the server and wait for its response. This step could be skipped completely.

It would also be interesting to do benchmarks with even more concurrent clients to see if this increases the efficiency of the optimizations even further. Varying the number of data servers and metadata servers could also prove to be interesting, because the metadata optimizations

reduce the load on the metadata servers. This load reduction is simply due to the fact that there are no more metafile objects, which would otherwise be managed by the metadata servers.

Also, the actual implementation is based on the modified version of PVFS from [Kun07]. This version in turn is based on the last official PVFS release version, which is already some months old. The reason for this is that the modified version offers enhanced tracing capabilities used for visualization. To enable wider testing or even integration into PVFS the implementation would need to be updated to the current development version, which features a significant number of changes.

The inefficiencies identified in chapter 6 could be eliminated to further improve metadata performance slightly. However, since these are minor inefficiencies the performance gain probably would be negligible.

Because the size of a file is not stored explicitly, but rather computed on the fly, metadata performance is not optimal if the file size is requested frequently. Since the default striping size is set to a mere 64 KByte, even a small file of size 1 MByte is striped across 16 data servers, if available.[1] So to compute the size of this file, 16 data servers have to be contacted, resulting in 16 message pairs. For use cases with read-mostly access patterns caching of the file size could thus greatly improve metadata performance.

[1] $16 \cdot 64\,\text{KByte} = 1\,\text{MByte}$

A Usage Instructions

In this appendix all necessary commands are given to compile, install, configure and use the modified PVFS version developed in this thesis. Additionally, instructions on how to run the benchmarks and create the visualization traces are given.

A.1 Installation of PVFS

Modified PVFS

The following commands install the modified version of PVFS used for the benchmarks in chapter 5.

```
$ tar xvjf pvfs-no_metafile.tar.bz2
$ cd pvfs-no_metafile
$ mkdir build install
$ cd build
$ ../configure --prefix="${PWD}/../install"
$ make
$ make install
$ cd ../install
$ export PATH="${PWD}/bin:${PWD}/sbin:${PATH}"
```

Modified PIOViz

The following commands install a full PIOViz environment – using the modified version of PVFS – as used for the visualization in chapter 6.

```
$ tar xvjf pioviz-no_metafile.tar.bz2
$ cd pioviz-no_metafile
$ mkdir install
$ echo "PREFIX=\"${PWD}/install\"" >> config.rc
$ make env
$ cd install
$ export PATH="${PWD}/bin:${PWD}/sbin:${PATH}"
```

A.2 Configuration of PVFS

The following commands configure PVFS in such a way that it may be used for benchmarks and visualization.

```
$ cd pvfs-no_metafile/no_metafile
$ pvfs2-genconfig --protocol tcp --tcpport 6666 \
```

```
  --ioservers localhost --metaservers localhost \
  --storage "${PWD}/pvfs2" --logfile "${PWD}/pvfs2.log" \
  --quiet fs.conf server.conf
$ sed -i -e "/^<\/Defaults>$/i \\\\tMpeLogFile ${PWD}/pvfs2" fs.conf
$ echo tcp://localhost:6666/pvfs2-fs /pvfs2 pvfs2 defaults,noauto 0 0 \
  > pvfs2tab
```

A.3 Starting PVFS

```
$ killall pvfs2-server
$ pvfs2-server fs.conf server.conf-localhost -r
$ pvfs2-server fs.conf server.conf-localhost -f
$ pvfs2-server fs.conf server.conf-localhost
```

A.4 Running the Benchmark

It is advisable to restart the PVFS server between each run of the benchmark. This can be done as described in appendix A.3.

The source code of these shell scripts can be found in appendix B.

```
$ ./run-single-client.sh normal 1
$ ./run-single-client.sh no_metafile 1
$ ./run-multiple-clients.sh normal 1
$ ./run-multiple-clients.sh no_metafile 1
```

A.5 Creating Visualization Traces

The source code of this shell script can be found in appendix B.

```
$ ./run-visualization.sh normal 1
$ ./run-visualization.sh no_metafile 1
```

B Benchmark and Visualization Scripts

This appendix contains the source code of all benchmark and visualization scripts. The `pvfs2-benchmark` and `pvfs2-visualization` tools are needed to use them. These are packaged with the modified version of PVFS.

run-single-client.sh

The script from listing B.1 runs one benchmark client as used in chapter 5. It expects two arguments:

1. Mode of operation (`normal` or `no_metafile`)
2. Run identifier (used as part of the output file name to differentiate between different runs)

It first creates a directory within PVFS. If the `no_metafile` mode is used, the `no_metafile` directory hint is set on this directory. Afterwards the stresstest script (see listing B.4) is run to use this directory as its root directory. The output of the stresstest script is stored in a text file.

Listing B.1: Script to run a single client

```
1  #!/bin/bash
2
3  set -x
4
5  die ()
6  {
7          echo "$@" >&2
8          exit 1
9  }
10
11 usage ()
12 {
13         die "Usage: ${0##*/} mode run"
14 }
15
16 [ -z "${1}" ] && usage
17 [ -z "${2}" ] && usage
18
19 MODE="${1}"
20 RUN="${2}"
```

```
21
22 [ ! -f pvfs2tab ] && die "pvfs2tab not found"
23
24 ROOT="$(awk '{ print $2; }' pvfs2tab | head -n 1)"
25
26 [ -z "${ROOT}" ] && "pvfs2tab malformed"
27
28 pvfs2-mkdir "${ROOT}/dir"
29
30 [ "${MODE}" = "no_metafile" ] && pvfs2-xattr -s -k
     ↪ user.pvfs2.no_metafile -v 1 "${ROOT}/dir"
31
32 ./stresstest.sh "${ROOT}/dir" 2> "pvfs-${MODE}-${RUN}.txt"
```

run-multiple-clients.sh

The script from listing B.2 runs five concurrent benchmark clients as used in chapter 5. It expects two arguments:

1. Mode of operation (`normal` or `no_metafile`)
2. Run identifier (used as a part of the output file name to differentiate between different runs)

It first creates five directories within PVFS. If the `no_metafile` mode is used, the `no_metafile` directory hint is set on each of these directories. Afterwards the stresstest script (see listing B.4) is run for each directory to use each it as its root directory. This script is run in the background such that all five stresstests run in parallel. The output of the stresstest scripts is stored in text files.

Listing B.2: Script to run five clients concurrently

```
1  #!/bin/bash
2
3  set -x
4
5  die ()
6  {
7          echo "$@" >&2
8          exit 1
9  }
10
11 usage ()
12 {
13         die "Usage: ${0##*/} mode run"
14 }
15
16 [ -z "${1}" ] && usage
17 [ -z "${2}" ] && usage
18
19 MODE="${1}"
20 RUN="{2}"
21
22 [ ! -f pvfs2tab ] && die "pvfs2tab not found"
23
24 ROOT="$(awk '{ print $2; }' pvfs2tab | head -n 1)"
25
26 [ -z "${ROOT}" ] && die "pvfs2tab malformed"
27
28 pvfs2-mkdir "${ROOT}/dira" "${ROOT}/dirb" "${ROOT}/dirc"
     ↪ "${ROOT}/dird" "${ROOT}/dire"
29
```

```
30 [ "${MODE}" = "no_metafile" ] && pvfs2-xattr -s -k
     ↪ user.pvfs2.no_metafile -v 1 "${ROOT}/dira"
31 [ "${MODE}" = "no_metafile" ] && pvfs2-xattr -s -k
     ↪ user.pvfs2.no_metafile -v 1 "${ROOT}/dirb"
32 [ "${MODE}" = "no_metafile" ] && pvfs2-xattr -s -k
     ↪ user.pvfs2.no_metafile -v 1 "${ROOT}/dirc"
33 [ "${MODE}" = "no_metafile" ] && pvfs2-xattr -s -k
     ↪ user.pvfs2.no_metafile -v 1 "${ROOT}/dird"
34 [ "${MODE}" = "no_metafile" ] && pvfs2-xattr -s -k
     ↪ user.pvfs2.no_metafile -v 1 "${ROOT}/dire"
35
36 ./stresstest.sh "${ROOT}/dira" 2> "pvfs-${MODE}-${RUN}-1.txt" &
37 ./stresstest.sh "${ROOT}/dirb" 2> "pvfs-${MODE}-${RUN}-2.txt" &
38 ./stresstest.sh "${ROOT}/dirc" 2> "pvfs-${MODE}-${RUN}-3.txt" &
39 ./stresstest.sh "${ROOT}/dird" 2> "pvfs-${MODE}-${RUN}-4.txt" &
40 ./stresstest.sh "${ROOT}/dire" 2> "pvfs-${MODE}-${RUN}-5.txt" &
41
42 wait
```

run-visualization.sh

The script from listing B.3 runs one visualization client as used in chapter 6. It expects two arguments:

1. Mode of operation (`normal` or `no_metafile`)
2. Run identifier (usually a number from an increasing sequence)

It first creates a directory within PVFS. If the `no_metafile` mode is used, the `no_metafile` directory hint is set on this directory. Afterwards the visualization script (see listing B.5) is run to use this directory as its root directory. The output of the visualization script is stored in a text file.

Listing B.3: Script to run the visualization script

```
1  #!/bin/bash
2
3  set -x
4
5  die ()
6  {
7          echo "$@" >&2
8          exit 1
9  }
10
11 usage ()
12 {
13         die "Usage: ${0##*/} mode run"
14 }
15
16 [ -z "${1}" ] && usage
17 [ -z "${2}" ] && usage
18
19 MODE="${1}"
20 RUN="${2}"
21
22 [ ! -f pvfs2tab ] && die "pvfs2tab not found"
23
24 ROOT="$(awk '{ print $2; }' pvfs2tab | head -n 1)"
25
26 [ -z "${ROOT}" ] && die "pvfs2tab malformed"
27
28 pvfs2-mkdir "${ROOT}/dir"
29
30 [ "${MODE}" = "no_metafile" ] && pvfs2-xattr -s -k
     ↪ user.pvfs2.no_metafile -v 1 "${ROOT}/dir"
31
32 ./visualization.sh "${ROOT}/dir" 2> "pvfs-${MODE}-${RUN}.txt"
```

stresstest.sh

The script from listing B.4 runs different tests to measure the performance of the three modified file system operations as shown in chapter 5. It expects one argument:

1. Root directory

In the first step, 100 directories are created. This is done outside of the actual benchmark, because it is not relevant. Afterwards each of these directories is populated with 500 empty files, which are then listed and finally removed. For each of these three operations the time needed is measured. At the end all directories are removed. Again, this is done outside of the actual benchmark.

Listing B.4: Stresstest script

```
1  #!/bin/bash
2
3  usage ()
4  {
5          echo "Usage: ${0##*/} root"
6          exit 1
7  }
8
9  [ -z "${1}" ] && usage
10
11 ROOT="${1}"
12
13 MKDIR="pvfs2-stresstest-mkdir"
14 TOUCH="pvfs2-stresstest-touch"
15 LS="pvfs2-stresstest-ls"
16 RM="pvfs2-stresstest-rm"
17 REAL_RM="pvfs2-rm"
18
19 list_files ()
20 {
21         echo "Listing files ..." >&2
22
23         "${LS}" "${ROOT}"
24
25         echo >&2
26 }
27
28 list_files_long ()
29 {
30         echo "Listing files (long) ..." >&2
31
32         "${LS}" -l "${ROOT}"
33
```

```
34          echo >&2
35  }
36
37  create_dirs ()
38  {
39          echo "Creating dirs ..." >&2
40
41          "${MKDIR}" "${ROOT}"
42
43          echo >&2
44  }
45
46  create_files ()
47  {
48          echo "Creating files ..." >&2
49
50          "${TOUCH}" "${ROOT}"
51
52          echo >&2
53  }
54
55  remove_dirs ()
56  {
57          echo "Removing dirs ..." >&2
58
59          for ((i = 0; i < 100; i++))
60          do
61                  "${REAL_RM}" "${ROOT}/${i}"
62          done
63
64          echo >&2
65  }
66
67  remove_files ()
68  {
69          echo "Removing files ..." >&2
70
71          "${RM}" "${ROOT}"
72
73          echo >&2
74  }
75
76  create_dirs
77  time create_files
78  time list_files > /dev/null
79  time list_files_long > /dev/null
80  time remove_files
```

```
81 remove_dirs
```

visualization.sh

The script from listing B.5 creates visualization traces for the three modified file system operations as shown in chapter 6. It expects one argument:

1. Root directory

Only one file is created, listed and finally removed. Before doing each of these three operations the tracing of server activity is enabled. Afterwards the tracing is deactivated and the log file sent through several processing steps.

Listing B.5: Visualization script

```
1  #!/bin/bash
2
3  usage ()
4  {
5          echo "Usage: ${0##*/} root"
6          exit 1
7  }
8
9  [ -z "${1}" ] && usage
10
11 ROOT="${1}"
12
13 TOUCH="pvfs2-visualization-touch"
14 LS="pvfs2-visualization-ls"
15 RM="pvfs2-visualization-rm"
16
17 trace_begin ()
18 {
19         pvfs2-set-eventmask -m "${ROOT}" -a all -o all
20 }
21
22 trace_end ()
23 {
24         OPERATION="${1}"
25
26         pvfs2-set-eventmask -m "${ROOT}" -a none -o none
27
28         mv "/tmp/pvfs2.clog2" "${OPERATION}.clog2"
29
30         clog2TOslog2 "${OPERATION}.clog2"
31
32         ProcessToGradient -o "${OPERATION}-final.slog2" -g "PC:.*"
               ↪ -m zero "${OPERATION}.slog2"
33         mv "${OPERATION}-final.slog2" "${OPERATION}-tmp.slog2"
```

```
34          EventToState -o "${OPERATION}-final.slog2" -j
                ↪ "start=SM-State(start);end=SM-State(end);
                ↪ final=SM-State;join=rank;join=cid;join=smp" -j
                ↪ "start=BMI (start);end=BMI
                ↪ (end);final=BMI;join=op;join=jid" -j "start=Job
                ↪ (start);end=Job (end);final=Job;join=op;join=jid" -j
                ↪ "start=Trove write (start);end=Trove write
                ↪ (end);final=Trove write;join=op;join=jid" -j
                ↪ "start=Trove read (start);end=Trove read
                ↪ (end);final=Trove read;join=op;join=jid"
                ↪ "${OPERATION}-tmp.slog2"
35          mv "${OPERATION}-final.slog2" "${OPERATION}-tmp.slog2"
36          Slog2ToCompositeSlog2 -o "${OPERATION}-final.slog2" -idorder
                ↪ "${OPERATION}-tmp.slog2"
37          mv "${OPERATION}-final.slog2" "${OPERATION}-tmp.slog2"
38          Slog2ToCompositeSlog2 -o "${OPERATION}-final.slog2" -pcorder
                ↪ "${OPERATION}-tmp.slog2"
39          mv "${OPERATION}-final.slog2" "${OPERATION}-tmp.slog2"
40          CompositeSlog2ToLineIDMap -o "${OPERATION}-final.slog2"
                ↪ "${OPERATION}-tmp.slog2"
41          mv "${OPERATION}-final.slog2" "${OPERATION}-tmp.slog2"
42          Slog2ToArrowSlog2 -o "${OPERATION}-final.slog2"
                ↪ "${OPERATION}-tmp.slog2"
43          rm "${OPERATION}-tmp.slog2"
44  }
45
46  list_file ()
47  {
48          echo "Listing file ..." >&2
49
50          trace_begin
51
52          "${LS}" "${ROOT}"
53
54          trace_end ls
55
56          echo >&2
57  }
58
59  list_file_long ()
60  {
61          echo "Listing file (long) ..." >&2
62
63          trace_begin
64
65          "${LS}" -l "${ROOT}"
66
```

```
67          trace_end lsl
68
69          echo >&2
70  }
71
72  create_file ()
73  {
74          echo "Creating file ..." >&2
75
76          trace_begin
77
78          "${TOUCH}" "${ROOT}/file"
79
80          trace_end touch
81
82          echo >&2
83  }
84
85  remove_file ()
86  {
87          echo "Removing file ..." >&2
88
89          trace_begin
90
91          "${RM}" "${ROOT}/file"
92
93          trace_end rm
94
95          echo >&2
96  }
97
98  create_file
99  list_file > /dev/null
100 list_file_long > /dev/null
101 remove_file
```

List of Figures

Listings

Bibliography

[BMLX03] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. Efficient Metadata Management in Large Distributed Storage Systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, San Diego, CA, April 2003.

[DW07] Ananth Devulapalli and Pete Wyckoff. File Creation Strategies in a Distributed Metadata File System. In *Proceedings of IPDPS '07*, Long Beach, CA, March 2007.

[KKK+07] Stephan Krempel, Michael Kuhn, Julian Kunkel, Christian Lohse, and Thomas Ludwig. Analysis of the MPI-IO Optimization Levels, 2007.

[Kre06] Stephan Krempel. Tracing the Connections Between MPI-IO Calls and their Corresponding PVFS2 Disk Operations. Bachelor's Thesis, Ruprecht-Karls-Universität Heidelberg, March 2006.

[Kun06] Julian Kunkel. Performance Analysis of the PVFS2 Persistency Layer. Bachelor's Thesis, Ruprecht-Karls-Universität Heidelberg, February 2006.

[Kun07] Julian Kunkel. Towards Automatic Load Balancing of a Parallel File System with Subfile Based Migration. Master's Thesis, Ruprecht-Karls-Universität Heidelberg, 2007.

[Tea] PVFS Development Team. Parallel Virtual File System. `http://www.pvfs.org`.

[WPBM04] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic Metadata Management for Petabyte-scale File Systems. In *Proceedings of SC '04*, Pittsburgh, PA, November 2004.