Interface Specification for the Multimedia Integrated Parallel File System (*MiPFS*)

Jesus Carretero and Alok Choudhary

Technical Report No. CPDC-TR-9802-013

February 18, 1998

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

Interface Specification for the Multimedia Integrated Parallel File System (*MiPFS*) [1] *Jesus Carretero and Alok Choudhary*

Electrical and Computing Engineering
Northwestern University
jcarrete@fi.upm.es

Abstract

This document proposes the interface to *MiPFS*, a parallel file system intended to be used as a low-level platform to develop more complex I/O entities, like MPI, on top of it. High-perfomance and standard functionality can be provided in client libraries or run-time systems built on top of *MiPFS*. We propose a reduced interface to parallel file system that includes basic I/O operations with scather-gather addressing, file control operations, hints specification, and quality of service negotiation.

This interface is based on the idea that the user should execute I/O operations using the data types stored on each object, so that each file can be managed as an *typed object*. However, as we want to provide only mechanism, and not policies, *MiPFS* includes functions to manage fixed and variable length records, and their associated indexes. This approach, and the quality of service functionality, allows to use *MiPFS* as a continuous media parallel file system.

Keywords: Parallel I/O systems, parallel file systems, multiprocessors, quality of service

[1] Partially supported by NATO Science Fellowships Program

Contents

Chapter 1

Introduction

This document describes the file system interface of the *MiPFS* parallel file system. Specifically, it includes functions to support reading and writing with scather-gather addressing for memory and file ranges, asynchronous operations, and some basic operations that can be performed on parallel files. The *MiPFS* interface is record oriented and not byte oriented. Thus, functions to manage file indexes and quality of service have been included in the mipfs interface to make the parallel file system suitable for applications using variable length records and continuous media files.

1.1 Document Notation and Conventions

All *MiPFS* names have an **MiPFS_** or **mipfs_** prefix, defined constants are in all capital letters, defined types have a **_t** extension, and variables have the first capital letter of each word (e.g. **FileMode**.)

Programs must not declare variables or functions with names beginning with the prefixes **MiPFS_** or **mipfs_**.

1.1.1 Interface Specification

Each function of the interface has been specified following the format of a manual page. Thus, each function has, at least, the following items:

NAME of the function and brief description.

SYNOPSIS of the function prototype.

PARAMETERS of the function, and their meaning.

DESCRIPTION of the function behavior and effects.

FILES of the interface related with the function described.

RETURNS codes (success or error code).

ERRORS codes got in case of failure.

SEE other functions related with this.

AUTHOR of the function.

Optionally, it might also have the following items:

WARNINGS .

BUGS existing in the current implementation.

COPYRIGHT .

RELEASE .

1.1.2 C Binding

Main conventions are:

- We use the ANSI C declaration format.
- The definition of named constants, function prototypes, and type definitions is supplied in an include file **mipfs_fs.h**.
- All C functions return an error code. The successful return code will always be MiPFS_SUCCESS. Failure return codes are also defined in this document. Almost all of them start with the prefixes **MiPFS_ERR_** or **MiPFS_INVALID_**.
- Type declarations are provided for handles to each category of opaque objects.
- Array arguments are indexed from zero.
- Logical options are unsigned integers with positive value. Its meaning is defined in the file **mipfs_constants.h**.
- File displacements are of type mipfs_offset_t, which is defined to be an unsigned integer of the size needed to hold any valid file size on the target architecture.

1.2 Organization of this Document

Chapter 2 highlights some design concepts, motivations, and decisions. It also provides a brief description of the overall parallel file system interface.

Chapter 3 defines the data types used in the interface specification. Even when understanding them is very important, reading of this chapter can be relegated after reading chapters 4 i through 9.

Chapters 4 through 9 contain the definition of the interface functionality. Those a re the core chapters of the document. The user should understand clearly the concepts behind the definitions, and thus chapter 2 has to be read previously.

The appendixes included in this document describes the **include** files where the interface, its constants, and its data types are defined. Those appendixes can be read after understanding all the previous chapters.

Chapter 2

Rationale

Parallel machines are being used increasingly to solve large, I/O intensive applications in several different disciplines with very different performance, semantic, and timing requirements. To get the maximum performance and flexibility for each kind of application, an increasing number of run-time I/O libraries and parallel file system have been developed upon the last years, leading to a lack of a standard application-programming interface (API) for parallel I/O. Different APIs are supporting by vendors and research projects, but most of them have little more than the basic UNIX functions in common. Moreover, usually, each parallel file system relays on a different parallel file structure an access model, which is incompatible with others. The result of those disparities is a lack of application's portability, and a lack of consensus within the parallel-IO and the applications community on a standard API.

In this chapter, some design hints of the *MiPFS* interface are provided. Those hints were carefully selected after reviewing previous work about parallel file system interfaces.

2.1 Background

Examples of existing commercial parallel file systems are IBM PIOFS [11] and Intel PFS [17]. The first relays on the concept of dynamic partitioning and subfiles, while the second has a POSIX-like interface extended with a set of file mode to define several access semantics. Examples of research parallel file systems are ParFiSys [7], PPFS [16], HFS [22], Galley [26], and PIOUS [23]. All of them propose their own API. ParFiSys proposes a POSIX interface extended with some specific high-performance and cache consistency functions. PPFS proposes a record oriented interface with many facilities to manage system configuration. HFS proposes an incremental approach to build the file system. They propose an initial API supporting Unix I/O, Stdio, and a user interface library, named Alloc Stream Facility, with a richer access interface. Galley proposes a strided, batched, and nested interface to access parallel files, which are conceived as a set of subfiles, each one can be also conceived as a set of forks. All the former parallel file systems try to increase I/O performance, specially for scientific applications. However, the applications community usually requires a very specific functionality to optimally exploit the profile of the I/O system in each kind of application. Trying to satisfy the former requirement, a number of run-time I/O libraries with special APIs have been developed. PASSION [9] provides software support for high-performance parallel I/O at the compiler, runtime and file system levels. Panda [8] provides synchronized i/o of large multidimensional arrays on parallel and sequential platforms. Jovian [3] provides collective operations, strided accesses, and operation "coalescing". Other libraries developed are SOLAR [31], for out-of-core linear algebra computations, ChemIO [13], for computational Chemistry applications, and TPIE [32] providing high-level data-parallel access to vectors of data on disk. The former APIs do not support an object model. Systems as ELFS [20], SHORE [6], and FRIGATE [21], provides support for persistent objects and extensibility, trying to satisfy also the high-performance requirements of parallel computation. MPI-IO [24] is the only real initiative trying to standarize an interface for parallel I/O at the application-programming level. Its goal is to design a standard file interface for SPMD message-passing programs providing data-type definition to access the files, individual and shared pointers, and collective operations. This API can be implemented on any parallel file system, a low-level parallel file system interface ,as the one proposed in the SIO initiative [12], or an abstract-device interface [30].

The former parallel file systems and I/O libraries are specifically oriented to provide high-performance for text and scientific files. Common procedures to achieve this goal are:

- file partitioning,
- specific access methods, many of the strided, and
- parallel I/O operations.

In addition, some of them try to increase performance by providing:

- access to complex data structures,
- collective operations,
- control over physical layout of data, and
- system extensibility.

However, none of them provides real-time/multimedia I/O, and quality of service as a primary goal, even when some of them have been used to provide multimedia services [19]. Several file systems for continuous media have been proposed. Swift [5] was designed for storing and retrieving large objects like color video or visualization data at very high speed. SPIFFI [14] is a video-on-demand server for networks of workstations. Magic [29] integrates data striping and playback schedule to provide video-on-demand services. And a few file systems have been proposed trying to integrate multi-service functionalities with quality of service. EFS [2] is a simple extent-based file system that uses the same disk-layout and meta-data format than CMFS [18], but with a more general API. Symphony [28]. However, their interfaces are very poor to be efficient in high-performance computation applications, their mechanisms to manage quality of service are far from those required in systems like [25], [1], or [4], and the possibilities of enhancing performance through informed prefetching and caching [27] are almost null.

2.2 Design Hints

After studying the systems described in the former section, it was clear to the authors that there is a lack of a parallel file system satisfying the requirements of future I/O demands for large-scale computing. Following the strategic research directions given in [10] and [15], a parallel file system should provide virtual devices interfaces and new programmer abstractions, should exploit data types and access patterns, should provide sophisticated prefetching, caching and scheduling, should also provide quality of service negotiation, and it should also allow the application influence the behavior of the system through behavior models and hints. Some of those design features cannot not be fulfilled having a monolithic parallel file system. As we have seen before, this approach led to the application's developers to create their own libraries to achieve high-performance I/O for their logical abstractions. Our feeling is that there is a lack of virtual device providing parallel I/O to files, allowing to use those files as typed files, and providing also efficient support for several kind of applications, like scientific, databases, or real-time applications. This feeling led us to design the parallel file system interface shown in this document.

The overall design goals leading our work are the following:

- Providing a very powerful low-level interface to use the parallel file system as a virtual device allowing to unify different I/O techniques. The parallel file system is equivalent to a micro-kernel. Multiple user abstractions can be developed on top of it.
- Providing generic mechanisms to accommodate any programmer abstraction into the parallel file system.
- Providing mechanisms, not policies, to allow the applications to influence the parallel file system behavior through data layout, access patterns, and prefetching and caching hints specification.
- Providing multiple service classes, including scientific applications, general-purpose applications, quality of service, and real-time applications, as video-service.
- Increasing parallel file system portability, reducing the implementation dependent parts and hiding the details of the physical storage system.

To accomplish the former goals, the following technical approaches are used in our parallel file system:

- Virtual device I/O techniques:
  - No files modes or shared pointers provided to increase parallelism.
  - Synchronous and asynchronous transfers with scather-gather i transfers and batching of operations.
  - File labeling to allow the applications to define specific features of a file without maintaining more files or be involved with file metadata.
- Programmer's abstractions accommodation:
  - Generic file types (fixed and variable length records) are provided to accommodate almost any programmer abstraction.
  - For variable length record files, operations to maintain a file index are also supported.
- Parallel file system behavior definition:
  - A generic file control operation is supported to influence parallel file system behavior at file or system level.
  - Control and hint specification operations are provided to specify space allocation, preallocation size, data layout, file type, caching information and file access patterns.

- Multiple service in a parallel file system may lead to a disparity in performance and policies. Our file system does not provide any semantic related with each kind of service, but a set of mechanisms basic to many kind of services. Some of them have been related above. Other are quality of service negotiation primitives and some hints to specify collective I/O operations.

2.3 *MiPFS* Functionality

Considering the former design goals, the *MiPFS* interface described in this document includes the following functionality:

- Error management.
- Basic I/O operations: open, close, unlink, rename.
- Synchronous and asynchronous file I/O.
- File Control operations: layout, caching, and hints.
- File Index operations.
- Quality of service negotiation and commitment.

This functionality may be slightly changed depending on the feedback received by further users of the system.

Chapter 3

MiPFS Data Types

This chapter is specially devoted to explain the main data types used in the *MiPFS* interface to manage a set of objects. Only structures significant to the user are described. Most of them have ranges defined in appendix B.

Internal structures are described in other internal design documents.

3.1 Basic Types

To enhance portability, the most usual types are redefined in *MiPFS* to avoid compiler or architecture dependences spreading through the whole code. Those are the only types to be modified if the code is ported (if needed to modify something).

```
typedef int             mipfs_int_t;

typedef unsigned int    mipfs_uint_t;

typedef long            mipfs_long_t;

typedef unsigned long   mipfs_ulong_t;

typedef float           mipfs_float_t;

typedef double          mipfs_double_t;
```

In addition, the following general POSIX data types are used:

- File name described as const char *. Maximum name length: **MiPFS_PATH_MAX** for absolute path names, not including the terminating zero.

- Memory address described as void *.

3.2 Single Types

mipfs_fileid_t

```
typedef mipfs_uint_t mipfs_fileid_t;
```

Type used to specify the file descriptor returned as a result of a create or open operation. **MiPFS_OPEN_MAX** is the maximum number of files that a process may be open simultaneously.

mipfs_mode_t

```
typedef mipfs_uint_t mipfs_mode_t;
```

Type used as a set of bits to specify the mode of a file operation. Valid modes are **MiPFS_RDONLY**, **MiPFS_WRONLY**, **MiPFS_RDWR** to specify that the file is being open for reading, writing, or for reading and writing simultaneously. Other constants than can be used in addition to the formerly specified are: **MiPFS_CREAT**, open with file create, **MiPFS_EXCL**, refuse to create if it already exists, and **MiPFS_TRUNC**, open with truncation. Modes used when testing files accessibility are: **MiPFS_READ_OK**, **MiPFS_WRITE_OK**, **MiPFS_EXEC_OK**, and **MiPFS_EXIST_OK**. File modes are important when opening, creating and hinting the file.

mipfs_offset_t

```
typedef mipfs_uint_t     mipfs_offset_t;
```

Offset into a file. Always defined in records. If the record size is 1-byte, the standard byte-oriented offset is being used. Other ways, this offset always means record offset. The translation of the record offset to the byte offset is made internally. Its absolute value is in the range [0...MiPFS_MAX_OFFSET]

mipfs_return_t

```
typedef mipfs_uint_t     mipfs_return_t;
```

Type used by functions to return a result code. **MiPFS_SUCCESS**, whose value must be 0, denotes success. Standard error described in appendix A.

mipfs_size_t

```
typedef mipfs_uint_t     mipfs_size_t;
```

Type used to describe sizes of file, records and memory regions. Absolute value in the range [0...MiPFS_MAX_SIZE].

mipfs_transfer_len_t

```
typedef mipfs_uint_t     mipfs_transfer_len_t;
```

Type used to count the total number of records (bytes) transferred in I/O operations. Range in [0...MiPFS_MAX_TRANSFER_LEN].

mipfs_io_list_len_t

```
typedef mipfs_uint_t     mipfs_io_list_len_t;
```

Type used to specify the total number of I/O operations included (batched) in listed I/O synchronous or asynchronous call. Be aware, that a strided I/O operation is considered as a single operation by the file system. Range in [0...MiPFS_MAX_LISTIO_LEN].

mipfs_fcntl_options_t

```
typedef mipfs_uint_t    mipfs_fcntl_options_t;
```

Set of bits to specify the options of a control operation. **MiPFS_CTL_OPTIONAL**, **MiPFS_CTL_MANDATORY**

mipfs_fcntl_op_t

```
typedef mipfs_uint_t    mipfs_fcntl_op_t;
```

Set of bits to specify the control operation code.

mipfs_cntl_class_t

```
typedef mipfs_uint_t    mipfs_cntl_class_t;
```

This is an unsigned integral type which contains the class identifier of hints passed with the mipfs_hint function.

mipfs_hint_options_t

```
typedef mipfs_uint_t    mipfs_hint_options_t;
```

mipfs_count_t

```
typedef mipfs_uint_t    mipfs_count_t;
```

This is an unsigned integral type with the range [0...MiPFS_MAX_COUNT]. It is used to represent a quantity of objects (records, bytes, etc.)

mipfs_caching_mode_t

```
typedef mipfs_uint_t     mipfs_caching_mode_t;
```

Type used to specify control actions on the client cache. Values are specified in chapter 6.

mipfs_caching_policy_t

```
typedef mipfs_uint_t     mipfs_caching_policy_t;
```

Type used to specify the cache policy to be applied to a file on the server and client caches. Values are specified in chapter 6. Default caching policies are used for different kind of files: video, test, scientific data, ... Defined policies are **MiPFS_CACHING_POLICY_LRU** and **MiPFS_CACHING_POLICY_GIC**, which stands for Generalized Caching Interval.

mipfs_consistency_scope_t

```
typedef mipfs_uint_t    mipfs_consistency_scope_t;
```

Type used to specify the cache consistency unit to be applied to a file on the server and client caches. Valid scopes for consistency units are: **MiPFS_FILE_SCOPE**, **MiPFS_RECORD_SCOPE**, **MiPFS_FILE_BLOCK_SCOPE**, and **MiPFS_REGION_SCOPE**. "Consistency regions" are also meaning when used with the last kind of scope.

mipfs_layout_options_t

```
typedef mipfs_uint_t    mipfs_layout_options_t;
```

Defines a set of bits with are ORed to specify the operations executed on the layout control function.

mipfs_aio_options_t

```
typedef mipfs_uint_t    mipfs_aio_options_t;
```

mipfs_aio_options_t define whether mipfs_acontrol will block or not when asking for an async op. **MiPFS_AIO_WAIT** specify a blocking asynchronous I/O or status operation, while **MiPFS_AIO_TEST** specifies a non-blocking operation. Moreover, **MiPFS_AIO_CANCELL** allows the application to cancell all the operations specified in the handler.

mipfs_aio_handle_t

```
typedef mipfs_uint_t mipfs_aio_handle_t;
```

Type used to identify asynchronous I/O ops. Each process can only have a maximum of **MiPFS_AIO_MAX** outstanding asynchronous operations.

mipfs_hintid_t

```
typedef mipfs_uint_t mipfs_hintid_t;
```

Type used to specify the hint descriptor returned as a result of a control operation specifying a hint. **MiPFS_HINT_MAX** is the maximum number of hints that a file may have simultaneously active.

mipfs_file_type_options_t

```
typedef mipfs_uint_t    mipfs_file_type_options_t;
```

Set of bits to specify the options of a file type control operation.

mipfs_time_t

```
typedef mipfs_uint_t mipfs_time_t;
```

Type used to specify time intervals. Mostly used for QoS. Value limited by **MiPFS_TIME_MAX**.

3.3 I/O operations

mipfs_io_strided_region_t

```
typedef struct {
       mipfs_offset_t          offset;
       mipfs_size_t            size;
       mipfs_size_t            stride;
} mipfs_io_strided_region_t;
```

Type used to specify generic strided regions on files. The region starts at "offset", it has a defined "size", and it is repeated with a certain "stride". Be aware that the number of iterations of the i/o region is not defined here, only its layout.

mipfs_mem_strided_region_t

```
typedef struct {
       void *addr;
       mipfs_size_t      size;
       mipfs_size_t      stride;
} mipfs_mem_strided_region_t;
```

Type used to specify generic strided regions on memory. The region starts at "addr", it has a defined "size", and it is repeated with a certain "stride". Be aware that the number of iterations of the memory region is not defined here, only its layout.

mipfs_file_io_list_t

```
typedef struct mipfs_file_io_list{
mipfs_io_strided_region_t io_region;
        mipfs_count_t            iterations;
        mipfs_io_list_len_t      file_io_vec_elem;
        struct mipfs_file_io_list  *file_io_vec;
} mipfs_file_io_list_t;
```

This structure is used to describe a collection of regions within a file that is involved in a parallel file system operation. Its purpose is to encapsulate the description of many simple transfers into one larger and more complex transfer to enable the file system to be more efficient in the execution of the total transfer. It can include only one file i/o list, for single operations, or a batched list to define a vector of batched memory positions. The field file_io_vec_elem means the number of operations batched.

mipfs_mem_io_list_t

```
typedef struct mipfs_mem_io_list{
        mipfs_mem_strided_region_t mem_region;
        mipfs_count_t             iterations;
        mipfs_io_list_len_t       mem_io_vec_elem;
        struct mipfs_mem_io_list  *mem_io_vec;
} mipfs_mem_io_list_t;
```

This structure is used to describe a collection of regions within a memory space that are involved in a parallel file system operation. Its purpose is to encapsulate the description of many simple transfers into one larger and more complex transfer to enable the file system to be more efficient in the execution of the total transfer. It can include only one memory list, for single operations, or a batched list to define a vector of batched memory positions. The field mem_io_vec_elem means the number of operations batched.

3.3.1 Asynchronous operations

mipfs_aio_control_t

```
typedef struct{
        mipfs_transfer_len_t     count;
        mipfs_return_t           status;
} mipfs_aio_control_t;
```

mipfs_aio_control_t, type used to return the status of an asynchronous I/O op. successful operation: count is set to the number of bytes transferred, and status to MiPFS_SUCCESS. On error status is set to a value indicating the nature of the error, and count is set to the number of bytes transferred correctly

3.4 Hint Specification

mipfs_hint_t

```
typedef struct{
        mipfs_hint_options_t     access_mode;
        mipfs_hint_options_t     access_pattern;
        mipfs_hint_options_t     hint_scope;
        mipfs_time_t      compute_time_between_IO;
        mipfs_time_t      time_to_reaccess;
        mipfs_file_io_list_t     *hint_io_list;
        mipfs_io_list_len_t       hint_list_len;
        mipfs_hintid_t      *hint_id;
} mipfs_hint_t;
```

This structure is used to describe hint information to help the file system to be more efficient in the execution of the global transfers.

Valid access modes are **MiPFS_RDONLY**, **MiPFS_WRONLY**, and **MiPFS_RDWR**.

Valid access patterns are **MiPFS_SEQUENTIAL**, **MiPFS_REVERSE**, and **MiPFS_RANDOM**.

Valid scopes for hints are **MiPFS_FILE_SCOPE** and **MiPFS_REGION_SCOPE**.

Valid computation and time to reaccess values are any quantity defined in mseconds and into the range [0...MiPFS_TIME_MAX].

3.5 Control and File Information Types

mipfs_fcntl_t

```
typedef struct{
        mipfs_fcntl_op_t         op_code;
        void                    *op_data;
        mipfs_fcntl_options_t    options;
        mipfs_return_t           result;
} mipfs_fcntl_t;
```

Type used to store the information associated with a control operation. Operation code in op_code, an indication in options of what to do if the control cannot be performed, and a pointer to a data buffer if necessary in op_data. The result field indicate success or failure.

mipfs_label_t

```
typedef struct{
        mipfs_size_t     size;
        void            *data;
} mipfs_label_t;
```

This type is used to store a file label, which can contain application- managed descriptive information about its associated file. The data field points to a memory buffer size bytes long.

mipfs_file_layout_t

```
typedef struct{
        mipfs_count_t           stripe_base;
        mipfs_count_t           stripe_width;
        mipfs_size_t            stripe_unit;
} mipfs_file_layout_t;
```

The number of parallel storage devices over which the file's data are striped is contained in the stripe width field, while the non-negative contiguous bytes stored on a device is the unit of striping. The stripe width does not include any devices containing redundancy information. The stripe_base, specifies the storage device to start file stripping. The file is striped over stripe_width devices starting at stripe_base device with a stripe_unit.

mipfs_file_type_t

```
typedef struct{
        mipfs_uint_t               file_type;
        mipfs_file_type_options_t  options;
        mipfs_uint_t               file_record_type;
        union{
                mipfs_size_t             file_record_size;  /* in bytes */
                mipfs_size_t             index_record_size;
        } size;
} mipfs_file_type_t;
```

The unit of logical storage for the file's data is defined in records. file_type defines the kind of file being used. Types currently supported are **MiPFS_FILE_TYPE_AUDIO**, **MiPFS_FILE_TYPE_VIDEO**, **MiPFS_FILE_TYPE_TEXT**, and **MiPFS_FILE_TYPE_SCIENTIFIC**. Each kind of file has associated default parameters into the parallel file system. Those values can be overridden with control operations. mipfs_file_type_options_t includes several characterizations of the record shape and features, as defined before in the constants section. file_record_type distinguishes among the two main types of files included in the current definition of the interface: variable and fixed length records, **MiPFS_VLR_RECORD_TYPE** and **MiPFS_FLR_RECORD_TYPE** respectively. record_size defines the length of the size, when fixed size record are used. For performance purposes some applications may want to define a VARIABLE length file type with a defined record_size. This is not allowed. For the file system purposes, it should be defined as a FIXED record length file. Be aware, that choosing a VLR file type may have important effects on performance and parallelism, as MiPFS cannot assume anything about the offset of the records in such kind of files. For VLR, the field index_record_size defines the length of the index entry (see mipfs_index_t), with a minimum sizeof(mipfs_index_t). The application has to be sure that the user-defined data fits into the interval $[3 * sizeof(mipfs\_size\_t), index\_record\_size]$.

mipfs_caching_consistency_t

```
typedef struct {
        mipfs_consistency_scope_t        granularity;
        mipfs_io_strided_region_t    *region;
        mipfs_count_t              regions_no;
} mipfs_caching_consistency_t;
```

This structure is used to describe a collection of regions within a file to maintain cache consistency. Its purpose is to encapsulate the description of many simple stride regions into one larger and more complex region to enable the file system to be more efficient in the execution of the consistency policy.

3.6 File Index Types

mipfs_index_t

```
typedef struct{
        mipfs_size_t     record_id;
        mipfs_size_t     record_offset;
        mipfs_size_t     record_length;
        void            *data;
} mipfs_index_t;
```

This type is used to store a record index for variable length record files. Such an index must contain at least three elements: record id, record offset into the file, and record length. Those fields are the minimum information to access the file in a per-record basis.

It can also contain application-managed descriptive information about compression, dimensions of a matrix, etc. MiPFS only will provide the mechanisms to manage this data type. Semantic related with the information included in the user-defined part of the index, such as compression algorithms, is not included into the file system. The user-data field points to a memory buffer "size" bytes long. MiPFS needs to know the size of the index records to compute their position into the file metadata, where the index is stored. This "size" is defined when the operation SetFileMode.

Anyway, the user can always access the file in a per-byte basis, assuming that he is going to extract the records later.

mipfs_index_file_list_t

```
typedef struct mipfs_index_file_list{
        mipfs_io_strided_region_t index_elem;
        mipfs_count_t  iterations;
        mipfs_count_t             index_vec_elem;
        struct mipfs_index_file_list  *index_vec;
} mipfs_index_file_list_t;
```

This structure is used to describe a collection of index within a file space that are involved in a parallel file system indexing operation. Its purpose is to encapsulate the description of many simple index transfers into one larger and more complex transfer to enable the file system to be more efficient in the execution of index operations.

It can include only one index list, for single operations, or a batched list to define a vector of batched index records. The field index_vec_elem means the number of operations batched. The field "index_no" is equivalent to the offset of a file, as the index records must have fixed length. Thus, the strided region for indexed may be defined as index no, index length, and index stride.

mipfs_index_mem_list_t

```
typedef struct mipfs_index_mem_list{
        mipfs_mem_strided_region_t    index_elem;
        mipfs_count_t             iterations;
        mipfs_count_t             index_vec_elem;
        struct mipfs_index_mem_list  *index_vec;
} mipfs_index_mem_list_t;
```

This structure is used to describe a collection of index within a memory space that are involved in a parallel file system indexing operation. Its purpose is to encapsulate the description of many simple index transfers into one larger and more complex transfer to enable the file system to be more efficient in the execution of index operations. It can include only one index list, for single operations, or a batched list to define a vector of batched index records. The field index_vec_elem means the number of operations batched.

3.7 Quality of Service Types

mipfs_qos_t

```
typedef struct{
        mipfs_time_t     start_time;
        mipfs_time_t     end_time;
        mipfs_size_t     max_bandwidth;
        mipfs_time_t     period;
        mipfs_count_t    records_per_period;
        mipfs_uint_t     commited_qoS;
        mipfs_uint_t     percent_guaranteed;
} mipfs_qos_t;
```

The QoS requirements of an application are defined using the following type. The application can specify the StartTime for the quality contract (the initial display time for a continuous media file), the EndTime, and the Period of the requests. Aperiodic requests can be implemented setting the Period parameter value to -1. To allow the file system to compute the resources needed, the maximum bandwidth requirement is provided. This parameter is significant for periodic and aperiodic requests. Moreover, for periodic requests, the number of RecordsPerPeriod can be defined. This parameter is meaningless for aperiodic requests, whose contrast should be negotiated only based upon the MaxBandwidth parameter. However, it is very important to make statistical assumptions about periodic data flows. Be aware, that usually the size of the RecordsPerPeriod requested is smaller than the maximum requested, as we talking of variable length records. The CommitedQoS parameter returns the quality of service that could be got. Three possible values are returned: MiPFS_QOS_DETERMINISTIC Requested totally guaranteed. MiPFS_QOS_STATISTICAL Requested partially guaranteed, MiPFS_QOS_BESTEFFORT Requested bandwidth or time not guaranteed. In case of a MiPFS_QOS_STATISTICAL guarantee, the percent offered is returned in the PercentGuaranteed parameter, so that 100requested has been commited, as a 60failure because of time or bandwidth constrains. O

Chapter 4

Basic File Operations

mipfs_open mipfs_close mipfs_link mipfs_unlink mipfs_rename

NAME

mipfs_open — Open a parallel file

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t   mipfs_open(
        mipfs_fileid_t   *FileDescriptor,
        char       *Name,
        mipfs_mode_t     Mode,
        mipfs_fcntl_t    *ControlOps,
        mipfs_count_t    ControlOpCount
);
```

PARAMETERS

FileDescriptor	On success, this will contain the file descriptor of the newly opened file.
Name	The name of the file to open.
Mode	The mode used to open the file. At least one of MiPFS_RDONLY, MiPFS_WRONLY, or MiPFS_RDWR. MiPFS_CREATE, to create a new file
ControlOps	Array of control operations to be performed on the file during the open.
ControlOpCount	The number of operations in the array specified by ControlOps.

DESCRIPTION

Open a file for reading and/or writing. This function takes a logical filename, and produces a filede- scriptor which supports reading and/or writing, depending on the value of Mode.

As part of the operation of opening the file, mipfs_open performs the control operations described by ControlOps and ControlOpCount.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.

Error code on error.

ERRORS

- **MiPFS_ERR_ALREADY_EXISTS** Creating a file that already exists.
- **MiPFS_ERR_CONTROL_FAILED** At least one of the mandatory control operations would have failed.
- **MiPFS_ERR_CONTROLS_CLASH** Some of the mandatory control operations specified are incompatible with each other.
- **MiPFS_ERR_FILE_NOT_FOUND** The file not exist and create not specified.
- **MiPFS_ERR_INVALID_FILENAME** The Name parameter is not a legal filename.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the open to fail.
- **MiPFS_ERR_MAX_OPEN_EXCEEDED** Opening the file would result in the task having too much open file descriptors.

SEE ALSO

mipfs_close mipfs_fcntl

AUTHOR

J. Carretero

NAME

mipfs_close — Closes a parallel file

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t   mipfs_close(
        mipfs_fileid_t   FileDescriptor
);
```

PARAMETERS

FileDescriptor   The file descriptor of the parallel file to close.

DESCRIPTION

This function closes a file. All resources associated with having the file open are deallocated. Cached pending writes are synced to disk. Result of any asynchronous ops. pending unspecified. Unused preallocated space may be released.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_INVALID_DESCRIPTOR** The FileDescriptor parameter does not refer to a valid FileDescriptor.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the close to fail.

SEE ALSO

mipfs_open mipfs_fcntl

AUTHOR

J. Carretero

NAME

mipfs_link — Link to a parallel file

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t mipfs_link(
char *Source,
char *Destination
);
```

PARAMETERS

Source	The name of the existing file to be linked with.
Destination	The name of the linked file to be created.

DESCRIPTION

The mipfs_link system call creates a new link (directory entry) for the existing file. Source points to a path name naming an existing file. Destination points to a path name naming the new directory entry to be created.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_FILE_OPEN** The file is open and the implementation does not allow open files to be linked.
- **MiPFS_ERR_FILE_NOT_FOUND** The source file does not exist.
- **MiPFS_ERR_ALREADY_EXISTS** Link with a Destination that already exists.
- **MiPFS_ERR_INVALID_FILENAME** The Source or Destination parameter is not a legal filename.
- **MiPFS_ERR_MAX_LINK_EXCEEDED** Linking with the source file would result in the file having too much linked files.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the open to fail.

SEE ALSO

mipfs_unlink

AUTHOR

J. Carretero

NAME

mipfs_unlink — Deletes a parallel file

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t mipfs_unlink(
char *Name
);
```

PARAMETERS

Name The name of the file to unlink.

DESCRIPTION

Removes an unopened file from the file system, deallocating its space. Open files may not be removed, but it is implementation dependent.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_FILE_OPEN** The file is open and the implementation does not allow open files to be unlinked.
- **MiPFS_ERR_FILE_NOT_FOUND** The file does not exist.
- **MiPFS_ERR_INVALID_FILENAME** The Name parameter is not a legal filename.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the open to fail.

SEE ALSO

mipfs_link mipfs_close

AUTHOR

J. Carretero

NAME

mipfs_access — Determine accessibility of a file

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t mipfs_access(
char *Name,
mipfs_mode_t Mode
);
```

PARAMETERS

Name The name of the file to test.

Mode The mode to be tested the file.

DESCRIPTION

This function takes a logical filename, and allows to test for its existence or mode, without opening the file. Mode test are referred to the owner of the parallel file. Legal modes are:

- **MiPFS_READ_OK** Read access
- **MiPFS_WRITE_OK** Write access
- **MiPFS_EXEC_OK** Execute (search) access
- **MiPFS_EXIST_OK** Check existence of file

Do not change the permanent state of the file.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_FILE_NOT_FOUND** The file does not exist.
- **MiPFS_ERR_INCORRECT_MODE** The mode of the file specified is not accepted or is wrong.
- **MiPFS_ERR_INVALID_FILENAME** The Name parameter is not a legal filename.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the test to fail.

SEE ALSO

mipfs_open mipfs_link

AUTHOR

J. Carretero

NAME

mipfs_rename — Changes the name of a parallel file

SYNOPSIS

```
#include "mipfs_fs.h"


mipfs_return_t mipfs_rename(
char *OldName,
char *NewName
);
```

PARAMETERS

OldName	The current name of the file to rename.
NewName	The new name of the file to rename.

DESCRIPTION

This function changes the name of a file form OldName to NewName. Renaming open files may not be allowed. Implementation dependent.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_ALREADY_EXISTS** NewName already exists.
- **MiPFS_ERR_FILE_OPEN** The file OldName is open and the implementation does not allow renaming open files.
- **MiPFS_ERR_FILE_NOT_FOUND** The file not exist.
- **MiPFS_ERR_INVALID_FILENAME** At least one of the names is not legal.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.

SEE ALSO

mipfs_open

AUTHOR

J. Carretero

Chapter 5

File I/O Operations

mipfs_read mipfs_write
mipfs_aread mipfs_awrite mipfs_acontrol

5.1 Synchronous I/O Operations

NAME

mipfs_read mipfs_write — Transfer data synchronously to/from file and memory

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t   mipfs_read(
        mipfs_fileid_t            FileDescriptor,
        mipfs_file_io_list_t      *FileList,
        mipfs_mem_io_list_t       *MemoryList,
        mipfs_transfer_len_t      *TotalTransferred
);


mipfs_return_t   mipfs_write(
        mipfs_fileid_t            FileDescriptor,
        mipfs_file_io_list_t      *FileList,
        mipfs_mem_io_list_t       *MemoryList,
        mipfs_transfer_len_t      *TotalTransferred
);
```

PARAMETERS

FileDescriptor	File descriptor of an open file
FileList	File data to be read or written
MemoryList	Memory buffers to transfer data
TotalTransferred	Total number of bytes read or written

DESCRIPTION

This functions move data between a list of file locations and a list of memory locations. All I/O must be done to a single file, in the FileDescriptor argument.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_INCORRECT_MODE** The mode of the file descriptor does not permit the I/O.
- **MiPFS_ERR_INVALID_FILE_DESCRIPTOR** FileDescriptor not valid.
- **MiPFS_ERR_INVALID_FILE_LIST** File regions described are invalid.
- **MiPFS_ERR_INVALID_MEMORY_LIST** Memory regions described are invalid.
- **MiPFS_ERR_UNEQUAL_LISTS** Different number of bytes in memory and file list
- **MiPFS_ERR_RECORD_LOST** Some guaranteed records have been lost.
- **MiPFS_ERR_JITTER** Maximum jittering specified in QoS negotiation has been overpassed.
- **MiPFS_ERR_LOW_THROUGHPUT** The throughput is lower than that guaranteed.
- **MiPFS_ERR_DELAY** The deadline of one or more requests could not be satisfied.
- **MiPFS_ERR_DISCONNECTION** The stream related with the file guaranteed has suffered a disconnection. No further operations can be guaranteed.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.
- **MiPFS_ERR_NO_SPACE** The system ran out of space while extending the file.

SEE ALSO

mipfs_aread mipfs_awrite mipfs_fcntl

AUTHOR

J. Carretero

5.2 Asynchronous I/O Operations

NAME

mipfs_aread mipfs_awrite — Asynchronously transfer data to/from file and memory

```
#include "mipfs_fs.h"

mipfs_return_t   mipfs_aread(
        mipfs_fileid_t          FileDescriptor,
        mipfs_file_io_list_t    *FileList,
        mipfs_mem_io_list_t     *MemoryList,
        mipfs_aio_handle_t      *Handle
);


mipfs_return_t   mipfs_awrite(
        mipfs_fileid_t            FileDescriptor,
        mipfs_file_io_list_t      *FileList,
        mipfs_mem_io_list_t       *MemoryList,
        mipfs_aio_handle_t        *Handle
);
```

PARAMETERS

FileDescriptor	File descriptor of an open file
FileList	File data to be read or written
MemoryList	Memory buffers to transfer data
Handle	Handle returned by the operation as I/O op. id.

DESCRIPTION

This functions are similar to mipfs_read and mipfs_write, but they only queue operations for processing in the parallel file system. They also move data between a list of file locations and a list of memory locations. All I/O must be done to a single file, in the FileDescriptor argument.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_INCORRECT_MODE** The mode of the file descriptor does not permit the I/O.
- **MiPFS_ERR_INVALID_FILE_DESCRIPTOR** FileDescriptor not valid.
- **MiPFS_ERR_INVALID_FILE_LIST** File regions described are invalid.

- **MiPFS_ERR_INVALID_MEMORY_LIST** Memory regions described are invalid.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.
- **MiPFS_ERR_NO_SPACE** The system ran out of space while extending the file.
- **MiPFS_ERR_UNEQUAL_LISTS** Different number of bytes in memory and file list

SEE ALSO

mipfs_read mipfs_write mipfs_fcntl

AUTHOR

J. Carretero

NAME

mipfs_acontrol — Execute control operations on asynchronous I/O handlers.

```
#include "mipfs_fs.h"

mipfs_return_t   mipfs_acontrol(
        mipfs_aio_handle_t     *HandleList,
        mipfs_count_t          HandleListLength,
        mipfs_aio_control_t    *Status,
        mipfs_aio_options_t    Options
);
```

PARAMETERS

HandleList	An array of mipfs_aio_handle_t identifying the asynchronous I/Os for which control is desired.
HandleListLength	The number of elements in HandleList.
Status	Pointer to an mipfs_aio_control_t to be filled in.
Options	Determines whether or not the operation blocks or returns immediately.

DESCRIPTION

Executes control operations on the asynchronous I/O handlers specified in HandleList. Control operations allowed are:

- **MiPFS_AIO_STATUS**, to check the current status of the asynchronous I/O operations.
- **MiPFS_AIO_WAIT**, to wait until completion of the asynchronous I/O operations. If several are specified, the operation will return when ALL have been completed.
- **MiPFS_AIO_CANCELL**, to cancell the asynchronous I/O operations specified in the HandleList.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_INVALID_HANDLE** Some of the handlers are not valid.
- **MiPFS_ERR_INVALID_FILE_LIST** File regions described are invalid.
- **MiPFS_ERR_INVALID_MEMORY_LIST** Memory regions described are invalid.
- **MiPFS_ERR_IO_CANCELED** The I/O was cancelled without completing.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.
- **MiPFS_ERR_IO_IN_PROGRESS** The I/O is still in progress.

- **MiPFS_ERR_MIXED_COL_AND_AIO** The implementation does not support mixed collective and asynchornous handlers.
- **MiPFS_ERR_RECORD_LOST** Some guaranteed records have been lost.
- **MiPFS_ERR_JITTER** Maximum jittering specified in QoS negotiation has been overpassed.
- **MiPFS_ERR_LOW_THROUGHPUT** The throughput is lower than that guaranteed.
- **MiPFS_ERR_DELAY** The deadline of one or more requests could not be satisfied.
- **MiPFS_ERR_DISCONNECTION** The stream related with the file guaranteed has suffered a disconnection. No further operations can be guaranteed.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.
- **MiPFS_ERR_NO_SPACE** The system ran out of space while extending the file.

SEE ALSO

mipfs_aread mipfs_awrite mipfs_fcntl

AUTHOR

J. Carretero

Chapter 6

File Control Operations

All the control operations affects to the file specified in the mipfs_fcntl operation. Control operations can be issued to manage file size, file labeling, file caching, file type, and file access hints.

mipfs_fcntl

6.1 File Control Operations

NAME

mipfs_fcntl — Perform a set of control operations on a given file.

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t   mipfs_fcntl(
        mipfs_fileid_t    FileDescriptor,
        mipfs_cntl_class_t    CntlClass,
        mipfs_fcntl_t     *Ops,
        mipfs_count_t     OpCount
);
```

PARAMETERS

FileDescriptor	The file descriptor of the open parallel file on which to perform the control operations.
CntlClass	The class of control operation being issued. It can basically be a CTL operation or a HINT specification.
Ops	An array of control operations to be performed.
OpCount	Number of control operations in the array referenced by Ops.

DESCRIPTION

This function performs the set of control operations (specified by the Ops argument on the open file specified by the FileDescriptor argu- ment. Each control operation is either mandatory or optional, de- pending on the bits set in its options field. The control operations can also be used to issue a set of predictions about the future accesses of this task (hints), to optimize future accesses to the file.

The application must not assume any ordering on the execution of the controls in Ops. If the operation fails for any reason, then all the control operations in Ops are annulled.

Operations:

MiPFS_CTL_GetSize	MiPFS_CTL_SetSize
MiPFS_CTL_GetPreallocation	MiPFS_CTL_SetPreallocation
MiPFS_CTL_GetLayout	MiPFS_CTL_SetLayout
MiPFS_CTL_GetFileType	MiPFS_CTL_SetFileType
MiPFS_CTL_GetCachingMode	MiPFS_CTL_SetCachingMode
MiPFS_CTL_GetCachingPolicy	MiPFS_CTL_SetCachingPolicy
MiPFS_CTL_Propagate	MiPFS_CTL_Refresh
MiPFS_CTL_GetLabel	MiPFS_CTL_SetLabel
MiPFS_CTL_GetConsistencyUnit	MiPFS_CTL_SetConsistencyUnit
MiPFS_CTL_GetHint	MiPFS_CTL_SetHint
MiPFS_CTL_CancelHint	MiPFS_CTL_CancelAllHints

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS All mandatory control operations succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_INVALID_DESCRIPTOR** The FileDescriptor parameter is not a valid FileDescriptor.
- **MiPFS_ERR_FILE_NOT_FOUND** The specified file did not exist .
- **MiPFS_ERR_CONTROL_FAILED** At least one of the mandatory control operations failed.
- **MiPFS_ERR_CONTROLS_CLASH** Some of the mandatory control operations are incompatible with each other and cannot be performed together.
- **MiPFS_ERR_HINT_TYPES_CLASH** The class of this hint differs from another previously defined for the same file region.
- **MiPFS_ERR_INVALID_CLASS** HintClass specified an invalid hint class.

SEE ALSO

mipfs_open mipfs_access

AUTHOR

J. Carretero

6.1.1 Control Operations Description

NAME

MiPFS_CTL_GetSize MiPFS_CTL_SetSize — Get or set the file size.

PARAMETERS

*Size mipfs_offset_t

DESCRIPTION

Get or set the file size, using the "*Size" parameter. Setting the size of a file will result in file size increasing or decreasing. If "*Size" is larger than the actual size, the file will be enlarger. However, if "*Size" is smaller than the actual size, the file will be TRUNCATED to that size, loosing the data among "*Size" and the actual size. To execute the former operation, the file must be open for writing.

Thus, a "MiPFS_CTL_SetSize" operation tries to allocate all the storage needed to enlarge the file, or frees the storage not needed after truncation. These functions do not modify the file offset for any open file descriptions associated with the file. On successful completion, if the file size is changed, these functions will update the metadata of the file. Further accesses beyond the new size will fail.

Those operations must not be confused with the preallocation operations described below.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_INCORRECT_MODE** The file descriptor's mode not permit the op.
- **MiPFS_ERR_NO_SPACE** The system cannot increase the storage for the file.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.

SEE ALSO

mipfs_fcntl MiPFS_CTL_SetPreallocation

NAME

MiPFS_CTL_GetPreallocation MiPFS_CTL_SetPreallocation — Get or set the preallocation unit for the file.

PARAMETERS

*Size mipfs_offset_t

DESCRIPTION

Get or set the preallocation unit for the file, allowing the applications to set the optimum amount of storage allocated each time resources are need. This operation allows to allocate extents defined by the application. Its goal is to enhance performance by achieving data locality on devices and by reducing resources allocation time. Typical allocation unit in UNIX files is 1 block. However, larger allocation units are often convenient on scientific and continuous media files.

The "MiPFS_CTL_SetPreallocation" operation can be considered also as a hint. Thus, it is only advisory for the parallel file system, as this operation can not be satisfied or it can compromise fairness into the system. The parallel file system will assign different preallocation values to different kind of files. Those values are implementation dependent.

Implementation may choose to ignore MiPFS_CTL_SetPreallocation, returning MiPFS_ERR_OP_UNSUPPORTED.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_INCORRECT_MODE** The file descriptor's mode not permit the op.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.
- **MiPFS_ERR_NO_SPACE** The system cannot increase the storage for the file in the defined amount.
- **MiPFS_ERR_OP_UNSUPPORTED** This operation is not supported.

SEE ALSO

mipfs_fcntl MiPFS_CTL_SetSize MiPFS_CTL_SetFileType

NAME

MiPFS_CTL_GetLayout MiPFS_CTL_SetLayout — Set/get the file's layout

PARAMETERS

*FileLayout mipfs_layout_t

DESCRIPTION

Set/get the file's layout, using a pointer to mipfs_layout_t to storage the file's layout. Allow the layout of a file's data on the underlying storage system to be queried and modified.

The only layout offered in this version is a striped file system. The start unit, stripe width, and stripe unit can be defined for each file. This operation is only allowed when the file is created. Further open operations must used the layout defined when the file was created.

Further versions might include different layouts depending on the feedback received by the users. However, the authors feel that the default layout is the most general one.

Implementation may choose to ignore MiPFS_CTL_SetLayout, returning MiPFS_ERR_OP_UNSUPPORTED.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_OP_ONLY_AT_CREATE** The implementation only supports layout setting on file creation.
- **MiPFS_ERR_INCORRECT_MODE** The mode of the file descriptor does not support the operation.
- **MiPFS_ERR_OP_UNSUPPORTED** This operation is not supported.

SEE ALSO

mipfs_fcntl

NAME

MiPFS_CTL_GetFileType MiPFS_CTL_SetFileType

PARAMETERS

*FileType mipfs_file_type_t

DESCRIPTION

Get the size of the record logical unit, the type of file to manage (Fixed length or variable length records) and some options about file management. The record size, when fixed, is the granularity of file accesses under recoed access policies. Anyway, ALL the FILES CAN BE ACCESSED AS A BYTE-ORIENTED FILE, specifying it in the file_type parameter. The default record length of those files is, of course, 1 byte.

The unit of logical storage for the file's data is defined in records. file_type defines the kind of file being used. Types currently supported are **MiPFS_FILE_TYPE_AUDIO**, **MiPFS_FILE_TYPE_VIDEO**, **MiPFS_FILE_TYPE_TEXT**, and **MiPFS_FILE_TYPE_SCIENTIFIC**. Each kind of file has associated default parameters into the parallel file system. Those values can be overridden with control operations. mipfs_file_type_options_t includes several characterizations of the record shape and features, as defined before in the constants section. file_record_type distinguishes among the two main types of files included in the current definition of the interface: variable and fixed length records, **MiPFS_VLR_RECORD_TYPE** and **MiPFS_FLR_RECORD_TYPE** respectively. record_size defines the length of the size, when fixed size record are used. For performance purposes some applications may want to define a VARIABLE length file type with a defined record_size. This is not allowed. For the file system purposes, it should be defined as a FIXED record length file. Be aware, that choosing a VLR file type may have important effects on performance and parallelism, as MiPFS cannot assume anything about the offset of the records in such kind of files. For VLR, the field index_record_size defines the length of the index entry (see mipfs_index_t), with a minimum sizeof(mipfs_index_t). The application has to be sure that the user-defined data fits into the interval $[3 * sizeof(mipfs\_size\_t), index\_record\_size]$.

Implementation may choose to ignore MiPFS_CTL_SetLayout, returning MiPFS_ERR_OP_UNSUPPORTED.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_CONTROLS_CLASH** Some of the mandatory control operations specified are incompatible with each other.
- **MiPFS_ERR_OP_ONLY_AT_CREATE** The implementation only supports record setting on file creation.
- **MiPFS_ERR_INCORRECT_TYPE** The file type specified is not supported by the parallel file system.
- **MiPFS_ERR_OP_UNSUPPORTED** This operation is not supported.

SEE ALSO

mipfs_fcntl

NAME

MiPFS_CTL_GetLabel MiPFS_CTL_SetLabel — Get/set the label of a file

PARAMETERS

*FileLabel mipfs_label_t

DESCRIPTION

Allow the label of a file's data on the underlying storage system to be queried and modified. This avoid the use of file headers, as the information pointed by the mipfs_label_t is stored as metadata and not inside the file itself.

The label is not interpreted by the parallel file system, allowing the applications to include any kind of descriptive information into the label. Maximum label length is MiPFS_MAX_LABEL_LENGTH. The contents of the label must be filled by the application. If the valid information is smaller than the defined in the "FileLabel" parameter, the remainder memory is in an undeterminated state.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_INCORRECT_MODE** The mode of the file descriptor does not support the operation.
- **MiPFS_ERR_INVALID_LABEL** The length of the new label, or the got label, exceeds
- **MiPFS_MAX_LABEL_LENGTH**.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.
- **MiPFS_ERR_NO_SPACE** No storage space to increase the file space.

SEE ALSO

mipfs_fcntl

NAME

MiPFS_CTL_GetCachingMode MiPFS_CTL_SetCachingMode — Get or set the file's caching mode in the client cache

PARAMETERS

* Mode mipfs_caching_mode_t

DESCRIPTION

Get or set the file's caching mode in the client cache.

Client Cache existence and consistency mode is using MiPFS_CTL_SetCachingMode, using the following constants:

- **MiPFS_CACHING_DISABLED**
- **MiPFS_CACHING_ENABLED_NO_CONSISTENCY**
- **MiPFS_CACHING_ENABLED_STRONG_CONSISTENCY**
- **MiPFS_CACHING_ENABLED_WEAK_CONSISTENCY**

This operation is ADVISORY for the parallel file system, thus requested changes can be denied.

Implementation dependent. Changes disallowed if the file is open by another task in the same client node.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_INCORRECT_MODE** The file descriptor's mode not permit the op.
- **MiPFS_ERR_ONLY_AT_OPEN** Changes on open files not supported.
- **MiPFS_ERR_CONTROLS_CLASH** This control operation is incompatible with other formerly specified and cannot be performed.
- **MiPFS_ERR_OP_UNSUPPORTED** The system does not support this operation.

SEE ALSO

mipfs_fcntl MiPFS_CTL_Propagate MiPFS_CTL_Invalidate

NAME

MiPFS_CTL_GetCachingPolicy MiPFS_CTL_SetCachingPolicy — Get or set the file's caching policy

PARAMETERS

*Policy mipfs_caching_policy_t

DESCRIPTION

Get or set the caching policy to be applied to a file on the server and client caches. Default caching policies are used for different kind of files: video, test, scientific data, ... Moreover, the cache policy defined might be modified because of formerly defined hints, as time to reuse. Defined policies are:

- **MiPFS_CACHING_POLICY_LRU**
- **MiPFS_CACHING_POLICY_GIC**, which stands for Generalized Caching Interval.

This operation is ADVISORY for the parallel file system, thus requested changes can be denied.

Implementation dependent. Changes disallowed if the file is open by another task in the same client node.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_INCORRECT_MODE** The file descriptor's mode not permit the op.
- **MiPFS_ERR_ONLY_AT_OPEN** Changes on open files not supported.
- **MiPFS_ERR_CONTROLS_CLASH** This control operation is incompatible with other formerly specified and cannot be performed.
- **MiPFS_ERR_OP_UNSUPPORTED** The system does not support this operation.

SEE ALSO

mipfs_fcntl MiPFS_CTL_GetCachingMode MiPFS_CTL_SetCachingMode MiPFS_CTL_Propagate MiPFS_CTL_Invalidate

NAME

MiPFS_CTL_Propagate

PARAMETERS

*File_IOList mipfs_file_io_list_t

DESCRIPTION

Forces the parallel file system to make any data associated with FileDescriptor and described in File_IOList visible to other nodes in the system. The implementation of the propagate operation is undefined, but further accesses to the propagated data should always get the most up to date version.

This operation has only meaning when used with the **MiPFS_CACHING_ENABLED_WEAK_CONSISTENCY** cache mode. When **MiPFS_CACHING_ENABLED_STRONG_CONSISTENCY** is used, data are immediately propagated. When **MiPFS_CACHING_ENABLED_NO_CONSISTENCY** is used there is not data propagation. If **MiPFS_CACHING_DISABLED** was defined, this operation returns an error, informing that the application is using a wrong semantic.

This operation only applies to the dirty data of the defined io_list. Only data from the open file is propagated.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_INVALID_FILE_LIST** At least one pointer of the File_io_list is not valid.
- **MiPFS_ERR_CONTROLS_CLASH** This control operation is incompatible with other formerly specified and cannot be performed.

SEE ALSO

mipfs_fcntl MiPFS_CTL_SetCachingMode MiPFS_CTL_GetCachingMode

NAME

MiPFS_CTL_Invalidate

PARAMETERS

*File_IOList mipfs_file_io_list_t

DESCRIPTION

Inform the parallel file system that locally cached data are invalid. Future read operations should receive up to date data from the servers.

This operation has only meaning when used with the **MiPFS_CACHING_ENABLED_WEAK_CONSISTENCY** cache mode. When **MiPFS_CACHING_ENABLED_STRONG_CONSISTENCY** is used, data are always valid, so invalidation has no meaning. Also, when **MiPFS_CACHING_ENABLED_NO_CONSISTENCY** is used there is not data invalidation. If **MiPFS_CACHING_DISABLED** was defined, this operation returns an error, informing that the application is using a wrong semantic.

This operation applies to the all data of the defined io_list. Only data from the open file is invalidated.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_INVALID_FILE_LIST** At least one pointer of the File_IOList is not valid.
- **MiPFS_ERR_CONTROLS_CLASH** This control operation is incompatible with other formerly specified and cannot be performed.

SEE ALSO

mipfs_fcntl MiPFS_CTL_SetCachingMode MiPFS_CTL_GetCachingMode MiPFS_CTL_Propagate

NAME

MiPFS_CTL_GetConsistencyUnit MiPFS_CTL_SetConsistencyUnit — Get/set the value of the consistency unit for the file

PARAMETERS

*FileConsistencyUnit mipfs_caching_consistency_t

DESCRIPTION

Get/set the size of the cache consistency unit and a list of consistency regions for a file. It is the granularity of cache consistency under weak and strong caching policies. Be aware that choosing a small consistency unit size may have negative effects on performance if the cache consistency policy is not careful and the files have many shared records.

Using this control operation, the applications may define "consistency regions", a set of disjoint file regions that never are shared for writing. This mechanism avoids false sharing effects when a strided access pattern is used on a file accessed in RDWR mode. When several regions are defined, the consistency size of each one is its size.

Valid scopes for consistency units are: **MiPFS_FILE_SCOPE**, **MiPFS_RECORD_SCOPE**, **MiPFS_FILE_BLOCK_SCOPE**, and **MiPFS_REGION_SCOPE**. "Consistency regions" are also meaning when used with the last kind of scope.

This operation is ADVISORY for the parallel file system, thus requested changes can be denied. Moreover, as cache consistency requires internal information to be stored in the parallel file system, this operation can be also denied if the granularity is too small.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_INCORRECT_MODE** The file descriptor's mode not permit the op.
- **MiPFS_ERR_ONLY_AT_OPEN** Changes on open files not supported.
- **MiPFS_ERR_INVALID_FILE_LIST** File regions described are invalid.
- **MiPFS_ERR_CONTROLS_CLASH** This control operation is incompatible with other formerly specified and cannot be performed.
- **MiPFS_ERR_OP_UNSUPPORTED** The system does not support this operation.

SEE ALSO

mipfs_fcntl

NAME

MiPFS_CTL_GetHint MiPFS_CTL_SetHint — Get/set a hint for the file

PARAMETERS

*HintType mipfs_hint_t

DESCRIPTION

Get/set a hint for the file. Following the type definition given for mipfs_hint_t, several hints may be defined with a single operation:

- Valid access modes are **MiPFS_RDONLY**, **MiPFS_WRONLY**, and **MiPFS_RDWR**.
- Valid access patterns are **MiPFS_SEQUENTIAL**, **MiPFS_REVERSE**, and **MiPFS_RANDOM**.
- Valid scopes for hints are **MiPFS_FILE_SCOPE** and **MiPFS_REGION_SCOPE**.
- Valid computation and time to re-access values are any quantity defined in mseconds and into the range [0...MiPFS_TIME_MAX].

Several hints may be defined for the same file consecutively. All hints are considered in FIFO order and its effect is valid until modified by another hint specification. The effect of each hint depends on the scope specified. Those defined as **MiPFS_FILE_SCOPE** are effective until cancelation or file closing. Those defined as **MiPFS_REGION_SCOPE** are effective while the region defined in the "hint_io_list" is begin accessed. Consecutive hints with same scope and region secludes the effect of previous hints. "hint_io_list" has no meaning when FILE scope is defined.

As several users may define contradictory hints to be used on the same file simultaneously, the hints are always considered ADVISORY by the parallel file system, never MANDATORY.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_HINT_CLASH** Some of the hint operations specified are incompatible with each other.
- **MiPFS_ERR_HINT_OPTIONS** Some of the hint options specified are incompatible with the operations or are not supported.
- **MiPFS_ERR_OP_ONLY_AT_CREATE** The implementation only supports hint specification on file creation.

SEE ALSO

mipfs_fcntl

NAME

MiPFS_CTL_CancelHint MiPFS_CTL_CancelAllHints — Cancel defined hints

PARAMETERS

*HintType mipfs_hint_t

DESCRIPTION

Cancels hints previously defined for the file. MiPFS_CTL_CancelHint allows to specify the hint to be canceled using the "hint_id" returned when the hint was set. MiPFS_CTL_CancelAllHints cancels ALL hints defined for the file. Hints are automatically canceled by the parallel file system when the data transfer predicted by the hint has occurred, but only when the REGION scope was defined for those hints.

RETURN VALUE

- **MiPFS_SUCCESS** The operation succeeded.
- **MiPFS_ERR_HINT_NOT_FOUND** Some of the hint operations specified were not found or the "hint_id" was erroneous.
- **MiPFS_ERR_HINT_CLASH** Some of the hint operations specified are incompatible with each other.
- **MiPFS_ERR_HINT_OPTIONS** Some of the hint options specified are incompatible with the operations or are not supported.

SEE ALSO

mipfs_fcntl

6.2 Parallel File System Control Operations

NAME

mipfs_sync — Flushes dirty cache blocks to physical storage.

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t   mipfs_sync(
void
);
```

PARAMETERS

DESCRIPTION

Causes all information in memory that should be on disk to be written out. This includes modified file system meta-data and delayed block I/O.

It should be used by commands and programs that examine a file system, such as mipfs_fsck, mipfs_df, etc. It is mandatory before a parallel file system shutdown.

The writing, although scheduled, is not necessarily complete upon return from mipfs_sync.

In systems that in some way ensures that the disks are always in a consistent state, or which does not cache buffers, this operation can be reduced to a no-op.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The sync operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the sync operation to fail.

SEE ALSO

mipfs_fcntl

AUTHOR

J. Carretero

Chapter 7

File Index Operations

mipfs_get_index mipfs_set_index

NAME

mipfs_get_index mipfs_set_index — Set/get the index values of a variable length record file.

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t   mipfs_get_index(
        mipfs_fileid_t                FileDescriptor,
        mipfs_index_file_list_t       *IndexFileList,
        mipfs_index_mem_list_t        *IndexMemList,
        mipfs_transfer_len_t          *TotalTransferred
);

mipfs_return_t   mipfs_set_index(
        mipfs_fileid_t                FileDescriptor,
        mipfs_index_file_list_t       *IndexFileList,
        mipfs_index_mem_list_t        *IndexMemList,
        mipfs_transfer_len_t          *TotalTransferred
);
```

PARAMETERS

FileDescriptor	File descriptor of an open file.
IndexFileList	List of index locations into the file index.
IndexMemList	List of index locations into the application memory.
TotalTransferred	Total number of bytes set or got.

DESCRIPTION

This functions move data to/from a list of in-memory index record locations and the file metadata, providing a strided interface. All I/O must be done to a single open file, defined in the FileDescriptor argument. Be aware that usually the stride is different in the application and in the file metadata. Contiguous application records may be discontiguos into the metadata, or reverse.

Indexed files need to define the file type and the index record size. Is the user responsibility to enforce this boundary. Information exceeding the index record size will be ignored.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_INCORRECT_MODE** The mode of the file descriptor does not permit the index operation I/O. Probably it is a fixed length records file or a byte-oriented one.
- **MiPFS_ERR_INVALID_FILE_DESCRIPTOR** FileDescriptor not valid.
- **MiPFS_ERR_INDEX_LIST** Index records described are invalid or the number of elements in the list is wrong.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.
- **MiPFS_ERR_NO_SPACE** The system ran out of space while extending the file metadata.

SEE ALSO

mipfs_open mipfs_fcntl

AUTHOR

J. Carretero

Chapter 8

Quality of Service Functionality

mipfs_qos_negotiate mipfs_qos_sign

NAME

mipfs_qos_negotiate — Negotiate a quality of service contract for the file

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t   mipfs_qos_negotiate(
                mipfs_fileid_t   FileDescriptor,
                mipfs_qos_t      *QoSRequested,
                mipfs_qos_t      *QoSAccepted
);
```

PARAMETERS

FileDescriptor	Open file descriptor for which the resources are requested.
QoSRequested	QoS requested by the application.
QoSAccepted	QoS accepted and offered by the parallel file system.

DESCRIPTION

This function allows the application to negotiate a contract of QoS with the parallel file system. The application needs are defined in the QoSRequested parameter.

To allow the file system to compute the resource needed, the maximum bandwidth requirement is provided. This parameter is significant for periodic and aperiodic requests. Moreover, for periodic requests, the number of RecordsPerPeriod can be defined. This parameter is meaningless for aperiodic requests, whose contrast should be negotiated only based upon the MaxBandwidth parameter. However, it is very important to make statistical assumptions about periodic data flows. Be aware, that usually the size of the RecordsPerPeriod requested is smaller than the maximum requested, as we talking of variable length records.

The CommitedQoS parameter defines the quality of service as a percent of the one requested, and the possible variation to have an acceptable QoS. As an example, a percent parameter of 100the QoS requested must be satisfied, while a 60the requested may be lost with acceptable conditions. QoSAccepted parameter defines the QoS offered by the parallel file system. The field CommitedQoS defines the percent of QoS committed.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_INVALID_DESCRIPTOR** The FileDescriptor parameter does not refer to a valid file descriptor.
- **MiPFS_ERR_INVALID_QOS_PARAMETERS** Some fields of the QoS request are incorrect or incompatible with others.
- **MiPFS_ERR_TIMES_CLASH** The start time is lower than the end time specified are incompatible with each other.
- **MiPFS_ERR_FILE_NOT_OPEN** The file is not open.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.

SEE ALSO

mipfs_open mipfs_fcntl mipfs_qos_sign

AUTHOR

J. Carretero

NAME

mipfs_qos_sign — Accept or reject a negotiated quality of service contract.

SYNOPSIS

```
#include "mipfs_fs.h"

mipfs_return_t   mipfs_qos_sign(
                mipfs_fileid_t   FileDescriptor,
                mipfs_uint_t     QoSOperation,
                mipfs_qos_t      *QoSAccepted
);
```

PARAMETERS

FileDescriptor	Open file descriptor for which the resources are accepted.
QoSoperation	Defines the operation to be executed on the negotiated contract. Two possibilities: MiPFS_QOS_ACCEPT or MiPFS_QOS_REJECT.
QoSAccepted	QoS accepted and offered by the parallel file system.

DESCRIPTION

This function allows the application to accept a contract of QoS already accepted by the parallel file system.

QoSAccepted parameter should include the QoS offered by the parallel file system as a result of a mipfs_qos_negotiate. Once accepted (MiPFS_QOS_ACCEPT) the resources are committed. If the contract is not accepted (MiPFS_QOS_REJECT), or not accepted is received after some time, the parallel file system releases the resources committed.

FILES

mipfs_fs.h

RETURN VALUE

MiPFS_SUCCESS The operation succeeded.
Error code on error.

ERRORS

- **MiPFS_ERR_INVALID_DESCRIPTOR** The FileDescriptor parameter does not refer to a valid file descriptor.
- **MiPFS_ERR_QOS_DIFFERS** The QoS accepted is different that the QoS offered by the system.
- **MiPFS_ERR_INVALID_QOS_PARAMETERS** Some fields of the QoS request are incorrect or incompatible with others.
- **MiPFS_ERR_TIMES_CLASH** The start time is lower than the end time.
- **MiPFS_ERR_FILE_NOT_OPEN** The file is not open.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.

SEE ALSO

mipfs_close mipfs_fcntl mipfs_qos_negotiate

AUTHOR

J. Carretero

Chapter 9

Error management

Valid error codes are described in Appendix C.

NAME

mipfs_error_string — Translates a return code to a string.

SYNOPSIS

```
#include "mipfs_fs.h"

const char* mipfs_error_string(
        mipfs_return_t   Result
);
```

PARAMETERS

Result The return code to translate.

DESCRIPTION

This function translates a return code to a string. The string codes and descriptions can be found in mipfs_errno.h.

FILES

mipfs_fs.h mipfs_error.h

RETURN VALUE

Error string on success.
NULL on error, e.g. inexistent error code.

AUTHOR

J. Carretero

Chapter 10

References

[1] G. Coulson A. Campbell and D. Hutchinson. A quality of service architecture. *IEEE Computer Communications Review*, April 1994.

[2] P. Barham. *Devices in a Multi-Service Operating System.* PhD thesis, Churchill College , University of Cambridge, July 1996.

[3] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, Mississippi State, MS, October 1994. IEEE Computer Society Press.

[4] L. Herrwich C. Volg, L. Wolf and R. Wittig. HeiRAT: Quality of service management for distributed multimedia systems. *Multimedia Systems Journal*, April 1996.

[5] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: A storage architecture for large objects. Technical Report UCSC-CRL-89-04, U.C. Santa Cruz, 1990.

[6] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394. ACM Press, 1994.

[7] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. Performance increase mechanisms for parallel and distributed file systems. *Parallel Computing*, 23(4):525–542, June 1997.

[8] Y. Chen, M. Winslett, S. Kuo, Y. Cho, M. Subramaniam, and K. E. Seamons. Performance modeling for the Panda array I/O library. In *Proceedings of Supercomputing '96*. ACM Press and IEEE Computer Society Press, November 1996.

[9] Alok Choudhary, Rajesh Bordawekar, Sachin More, K. Sivaram, and Rajeev Thakur. PASSION runtime library for the Intel Paragon. In *Proceedings of the Intel Supercomputer User's Group Conference*, June 1995.

[10] Alok Choudhary and David Kotz. Large-scale file systems with the flexibility of databases. *ACM Computing Surveys*, 28A(4), December 1996. Position paper

for the Working Group on Storage I/O for Large-Scale Computing, ACM Workshop on Strategic Directions in Computing Research. Available on-line only, at http://www.acm.org/surveys/1996/ChoudharyFile/.

[11] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, January 1995.

[12] Peter F. Corbett, Jean-Pierre Prost, Chris Demetriou, Garth Gibson, Erik Reidel, Jim Zelenka, Yuqun Chen, Ed Felten, Kai Li, John Hartman, Larry Peterson, Brian Bershad, Alec Wolman, and Ruth Aydt. Proposal for a common parallel file system programming interface. WWW http://www.cs.arizona.edu/sio/api1.0.ps, September 1996. Version 1.0.

[13] Ian Foster and Jarek Nieplocha. ChemIO: High-performance I/O for computational chemistry applications. WWW http://www.mcs.anl.gov/chemio/, February 1996.

[14] Craig S. Freedman and David J. DeWitt. The SPIFFI scalable video-on-demand system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 352–363. ACM Press, 1995.

[15] Garth A. Gibson, Jeffrey Scott Vitter, and John Wilkes. Report of the working group on storage I/O issues in large-scale computing. *ACM Computing Surveys*, 28(4):779–793, December 1996.

[16] James V. Huber, Jr. PPFS: An experimental file system for high performance parallel input/output. Master's thesis, Department of Computer Science, University of Illinois at Urbana Champaign, February 1995.

[17] Intel Scalable Systems Division. *Paragon System User's Guide*, may 1995.

[18] P. Jardetzky. *Network File Server Design for Continuous Media*. PhD thesis, Computer Laboratory, University of Cambridge, October 1992. Also available as technical report No. 268.

[19] Edgar T. Kalns and Yarsun Hsu. Video on demand using the Vesta parallel file system. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 30–46, April 1995.

[20] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, Portland, OR, October 1994. ACM Press.

[21] T. Kim and G. Popek. Frigate: an object-oriented file system for ordinary users. In *Proc. of the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS'97)*, Portland, USA, June 1997.

[22] Orran Krieger and Michael Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 95–108, Philadelphia, May 1996. ACM Press.

[23] S. Moyer and V. S. Sunderam. Parallel I/O as a parallel application. *International Journal of Supercomputer Applications*, 9(2):95–107, Summer 1995.

[24] MPI-IO: a parallel file I/O interface for MPI. The MPI-IO Committee, April 1996. Version 0.5. See WWW http://lovelace.nas.nasa.gov/MPI-IO/mpi-io-report.0.5.ps.

[25] K. Narhstedt and J. Smith. The QoS broker. *IEEE Multimedia*, 2(1):53–67, April 1995.

[26] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[27] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[28] Sriram Rao Prashant Shenoy, Pawan Goyal and Harrick Vin. Symphony: An integrated multimedia file system. In *Proc. of the SPIE/ACM Conference on Multimedia Computing and Networking (MMCN'98)*, San Jose, CA, USA, 1998. Also available as Technical Report TR-97-09, Department of Computer Sciences, Univ. of Texas at Austin.

[29] Herb Taylor, Danny Chin, and Stan Knight. The Magic video-on-demand server and real-time simulation system. *IEEE Parallel and Distributed Technology*, 3(2):40–51, Summer 1995.

[30] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[31] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 28–40, Philadelphia, May 1996. ACM Press.

[32] Darren Erik Vengroff. A transparent parallel I/O environment. In *Proceedings of the 1994 DAGS/PC Symposium*, pages 117–134, Hanover, NH, July 1994. Dartmouth Institute for Advanced Graduate Studies.

Appendix A

Include Files

The following four include files are used to specify the constants, data types, and function prototypes used in *MiPFS* interface.

A.1 mipfs_fs.h

The file "mipfs_fs.h" contains the function declarations and/or prototypes for all functions defined in the file system interface. Applications or higher-level libraries must include "mipfs_fs.h" in their source before referencing any of the types, constants, or functions described in the API. Specifically, it includes functions to support reading and writing with scatter-gather addressing for memory and file ranges, asynchronous operations, and some basic operations that can be performed on parallel files.

As *MiPFS* interface is record oriented, functions to manage file indexes and quality of service have been included in the mipfs interface to make the parallel file system suitable for applications using variable length records and continuous media files.

The files "mipfs_types.h", "mipfs_constants.h", and "mipfs_error.h" are included in "mipfs_fs.h" to get the constants, data types, and error codes used in the interface specification.

A.2 mipfs_types.h

This file defines the data types which are referenced in the basic API. Most of them have ranges defined in the Range Constants file "mipfs_constants.h".

Three main kind of types are defined:

1. *MiPFS* basic types, used to define the single types.
2. *MiPFS* single types, used to define the complex types.
3. *MiPFS* complex types.

See chapter 3 for a detailed description of the data types.

A.3 mipfs_constants.h

This file describes the constants used in this basic API to specify both numerical values for logical names and the ranges of data types. These constants are implementation-specific, specially the range definition. However, for each of them, both a minimum value and a recommended value are given.

See appendix B for a detailed description of the data types.

A.4 mipfs_error.h

This file describes the error codes used in this basic API to return errors in the parallel file system operations. They define a numerical values for the logical error names. These constants are implementation-specific, however a value must be given to each of them.

The user should always use the names defined in this appendix, and never the implementation numerical value. The latest policy is error prone and greatly reduces portability.

See appendix C for a detailed description of the error codes.

Appendix B

MiPFS Constants

This appendix describes the constants used in this API to specify the ranges of some data types. Only constants significant to the user are described.

Be aware that those constants are implementation specific, thus the range of some of them will be different on each implementation. Anyway, a maximum and a minimum will be defined with the logical name introduced in this chapter. See file **mipfs_constants.h** for actual values and other constants.

B.1 General Purpose Constants

PUBLIC		Exported data and functions(default)
PRIVATE	static	Internal data and functions
MiPFS_SUCCESS	0	Success return code

B.2 Range Constants

MiPFS_MAX_LISTIO_LEN	256
MiPFS_MAX_COUNT	INT32_MAX
MiPFS_MAX_PATH_MAX	1024
MiPFS_MAX_LABEL_LEN	MiPFS_MAX_NAME_LEN
MiPFS_MAX_OFFSET	INT64_MAX
MiPFS_OPEN_MAX	512
MiPFS_MAX_SIZE	INT64_MAX
MiPFS_MAX_TRANSFER_LEN	MiPFS_MAX_SIZE
MiPFS_AIO_MAX	512
MiPFS_TIME_MAX	INT32_MAX
MiPFS_HINT_MAX	256

B.3 Options and Flags Constants

MiPFS_RDONLY MiPFS_WRONLY MiPFS_RDWR MiPFS_CREAT MiPFS_EXCL MiPFS_TRUNC

MiPFS_READ_OK MiPFS_WRITE_OK MiPFS_EXEC_OK MiPFS_EXIST_OK.

MiPFS_CTL_OPTIONAL MiPFS_CTL_MANDATORY

MiPFS_AIO_WAIT MiPFS_AIO_TEST MiPFS_AIO_CANCELL
MiPFS_VLR_RECORD_TYPE MiPFS_FLR_RECORD_TYPE
MiPFS_FILE_TYPE_AUDIO MiPFS_FILE_TYPE_VIDEO MiPFS_FILE_TYPE_TEXT MiPFS_FILE_TYPE_SCIENTIFIC
MiPFS_QOS_DETERMINISTIC MiPFS_QOS_STATISTICAL MiPFS_QOS_BESTEFFORT
MiPFS_SEQUENTIAL MiPFS_REVERSE MiPFS_RANDOM
MiPFS_FILE_SCOPE MiPFS_REGION_SCOPE MiPFS_RECORD_SCOPE MiPFS_FILE_BLOCK_SCOPE
MiPFS_CACHING_POLICY_LRU MiPFS_CACHING_POLICY_GIC
MiPFS_CACHING_DISABLED MiPFS_CACHING_ENABLED_NO_CONSISTENCY MiPFS_CACHING_ENABLED_STRONG_CONSISTENCY MiPFS_CACHING_ENABLED_WEAK_CONSISTENCY

Appendix C

MiPFS Error Codes

This appendix describes the error codes used in this API to specify the errors occurred while executing *MiPFS* operations. Only error codes returned by the interface functions are described.

Be aware that those error codes are implementation specific, thus the value of some of them will be different on each implementation. The user should always use the names defined in this appendix, and never the implementation numerical value. The latest policy is error prone and greatly reduces portability. See file **mipfs_error.h** for actual values.

- **MiPFS_ERR_ALREADY_EXISTS** Creating a file that already exists or NewName already exists.
- **MiPFS_ERR_CONTROLS_CLASH** Some of the mandatory control operations specified are incompatible with each other.
- **MiPFS_ERR_CONTROL_FAILED** At least one of the mandatory con trol operations would have failed.
- **MiPFS_ERR_DELAY** The deadline of one or more requests could not be satisfied.
- **MiPFS_ERR_FILE_NOT_FOUND** The file not exist and, when opening the file, create not specified.
- **MiPFS_ERR_FILE_OPEN** The file is open and the implementation does not allow renaming or unlinking open files.
- **MiPFS_ERR_INCORRECT_TYPE** The file type specified is not supported by the parallel file system.
- **MiPFS_ERR_HINT_NOT_FOUND** Some of the hint operations specified were not found or the "hint_id" was erroneous.
- **MiPFS_ERR_HINT_TYPES_CLASH** The class of this hint differs from another previously defined for the same file region.
- **MiPFS_ERR_HINT_CLASH** Some of the hint operations specified are incompatible with each other.
- **MiPFS_ERR_HINT_OPTIONS** Some of the hint options specified are incompatible with the operations or are not supported.

- **MiPFS_ERR_INCORRECT_MODE** The mode of the file descriptor does not permit the I/O or index operation. For the index operations, probably the file has fixed length records or is a byte-oriented file.
- **MiPFS_ERR_INDEX_LIST** Index records described are invalid or the number of elements in the list is wrong.
- **MiPFS_ERR_INVALID_CLASS** HintClass specified an invalid hint class.
- **MiPFS_ERR_INVALID_DESCRIPTOR** FileDescriptor does not refer to a valid file descriptor.
- **MiPFS_ERR_INVALID_FILENAME** At least one of the names is not a legal filename.
- **MiPFS_ERR_INVALID_FILE_DESCRIPTOR** FileDescriptor not valid.
- **MiPFS_ERR_INVALID_FILE_LIST** File regions described are invalid.
- **MiPFS_ERR_INVALID_FILE_NAME** The name given by FileName is invalid.
- **MiPFS_ERR_INVALID_HANDLE** Some of the handlers are not valid in an asynchronous operation.
- **MiPFS_ERR_INVALID_MEMORY_LIST** Memory regions described are invalid.
- **MiPFS_ERR_INVALID_QOS_PARAMETERS** Some fields of the QoS request are incorrect or incompatible with others.
- **MiPFS_ERR_IO_CANCELED** The I/O operation was canceled without completing.
- **MiPFS_ERR_IO_FAILED** A physical I/O error caused the operation to fail.
- **MiPFS_ERR_IO_IN_PROGRESS** The I/O operation is still in progress.
- **MiPFS_ERR_JITTER** Maximum jittering specified in QoS negotiation has been overpassed.
- **MiPFS_ERR_LOW_THROUGHPUT** The throughput is lower than that the one negotiated in the QoS contract.
- **MiPFS_ERR_MAX_OPEN_EXCEEDED** Opening the file would result in the task having too much open file descriptors.
- **MiPFS_ERR_MAX_LINK_EXCEEDED** Linking with the source file would result in the file having too much linked files.
- **MiPFS_ERR_MIXED_COL_AND_AIO** The implementation does not support mixed collective and asynchronous handlers.
- **MiPFS_ERR_NO_SPACE** The system ran out of space while extending the file data or metadata.
- **MiPFS_ERR_QOS_DIFFERS** The QoS accepted is different that the QoS offered by the system.

- **MiPFS_ERR_RECORD_LOST** Some guaranteed records have been lost.
- **MiPFS_ERR_TIMES_CLASH** The start time is lower than the end time specified are incompatible with each other.
- **MiPFS_ERR_UNEQUAL_LISTS** Different number of bytes in memory and file list.