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

Bachelorarbeit

Tracing Internal Behavior in PVFS

Name: Tien Duc Dinh
Matrikelnummer: 2592222
Betreuer: Julian M. Kunkel, Prof. Thomas Ludwig
Abgabe Datum: 5 October, 2009

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

.....

Datum: 05.10.2009

Abstract

Nowadays scientific computations are often performed on large cluster systems because of the high performance they deliver. In such systems there are many reasons for bottlenecks which are related to both hardware and software. This thesis defines and implements metrics and information used for tracing events in MPI applications in conjunction with the parallel file system PVFS in order to localize bottlenecks and determine system behavior. They are useful for the optimizations of the system or applications. After tracing, data is stored in trace files and can be analyzed via the visualization tool Sunshot.

There are two experiments made in this thesis. The first experiment is made on a balanced system. In this case Sunshot shows a balanced visualization between nodes, i.e. the load between nodes looks similar. Moreover, in connection with this experiment the new metrics and tracing information or characteristics are discussed in detail in Sunshot. In contrast, the second experiment is made on an unbalanced system. In this case Sunshot shows where bottlenecks occurred and components which are related.

Acknowledgments

So my first thanks go to Prof. Dr. Thomas Ludwig and Julian M. Kunkel for the guidance and support during the whole project period. I am also especially grateful to my friends - Christian Takacs and Vu, Duy Viet - for checking some english mistakes.

Last but not least, I want to appreciate the valuable support of my parents Cuong and Hang.

Contents

1 Introduction

1.1 Problem Statement

The Message Passing Interface (MPI) is an application programming interface that facilitates the development of parallel applications and libraries. MPI is designed to provide access to parallel hardware, e.g. clusters, heterogeneous networks or parallel computers with the goals of high performance, scalability and portability. Today it is widely used for high-performance computing.

MPI supports both point-to-point and collective communication, and so far there are now two models of MPI: MPI-1 and MPI-2. While the principal MPI-1 model defines only active communication routines, MPI-2, an extension of MPI-1, has a limited distributed shared memory concept and I/O concepts in form of MPI-IO which can be built on top of parallel file systems such as PVFS. In general MPI applications have parallel access to nodes across the cluster via a parallel file system. The system is very complex. Therefore, it is difficult for the users or developers to determine the problems or inefficient operating components if the system did not work properly. A solution is to determine the program behavior by tracing the parallel file system together with the (MPI) program to localize bottlenecks.

1.2 Related Work and State of the Art

For tracing data there are two concepts which are used for different purposes. First is the online tracing. In this concept we use a monitoring system to get data of the application and/or instrument this application. Data is not stored, instead it is immediately used for several purposes: either we can display it for run-time performance analysis, or control applications (e.g. with a load balancer). A well-known representative of this concept is Paradyn [Pagb]. Second is the offline concept. In comparison to the first concept, data will be stored as trace files consisting of program activities. After program completion (or with a considerable delay during program run) data is processed for further analysis. Representatives are TAU, the Intel Trace Analyzer, Vampir and PIOViz [LKK+07].

1.2.1 Vampir

Vampir is a tool developed at the University of Dresden. Its goal is to trace client I/O applications. It consists of a tool set which supports many trace aspects including MPI calls, POSIX IO semantics, POSIX Threads and OpenMPI on many platforms like Linux, Sun Solaris, IBM AIX (PPC), SGI IRIX (MIPS), Mac OS and Windows. This tool instruments (modifies) a given application in order to invoke additional measurement calls during runtime.

Vampir supports tracing of hardware performance counters and MPI calls. For performance counters it collects information of hardware via the PAPI library [1] (Unix/Linux), CPC library (Solaris). On NEC SX machines it uses special register calls to query the processor's hardware counters. Moreover, it uses the library **getrusage** to collect information about consumed resources and operating system events of processes (e.g. user/system time, received signals and context switches) and LIBC for memory allocation and free functions (e.g. malloc, realloc, free). For MPI calls Vampir traces some important information and communications (e.g. participating processes, transferred bytes, tag and communicators). For I/O Vampir traces LIBC I/O calls including file name and amount of transferred data [Pagc].

Another possible tracing environment is to use PIOViz which supports the tracing of MPI applications and PVFS. It is introduced in the next section.

1.2.2 PIOViz

The working group Parallel and distributed systems (PVS) provides PIOViz [LKK+06], an environment with MPICH2 and PVFS which allows to visualize activities on the parallel file system servers in conjunction with the client events triggering these activities. Provided features should support developers to optimize their MPI applications and to improve the parallel file system. In brief, the environment contains a set of user-space tools, modifications to the low level layer ADIO (part of the MPICH2's MPI-IO implementation) [TGL96], MPE [2] and logging enhancements to PVFS [Kun07].

PIOViz 1.0 Characteristics

PIOViz 1.0 uses MPICH, which includes MPE, an implementation for tracing client/server applications by linking the program with tracing libraries which intercept MPI calls. There is a single trace file on each client/server side after program completion. These two single files will be then merged into a single trace file via several intermediate tools and can be then visualized via Jumpshot (part of MPE). Jumpshot provides useful information of the system, arrows are shown for the correlation between client and server activities.

An example for correlations between client and server with independent/collective MPI calls on contiguous/non-contiguous data in PIOViz 1.0 is shown in figure 1.1. Note detail information about I/O size is provided upon mouse over of I/O operations. Some notes have been made directly on the screenshots.

[1] Performance Application Programming Interface
[2] MPI Parallel Environment, also known as Multi-Processing Environment

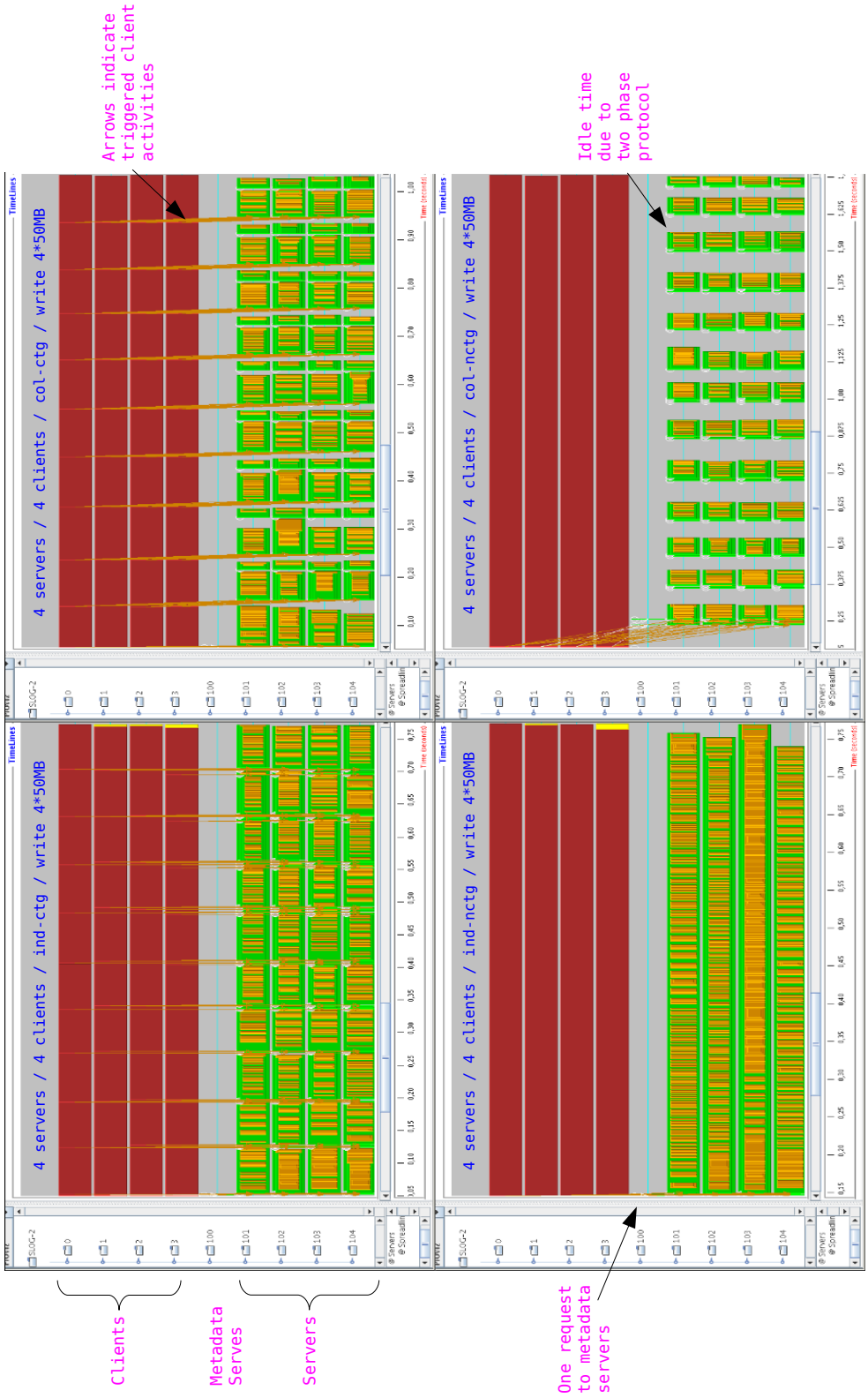

Figure 1.1: Correlation between client and server in PIOViz 1.0

Furthermore, PIOViz has provided some statistics assisting analysis. An excerpt is shown in table 1.1.

Name	Statistic provider
Average load for one minute	Linux kernel
Total memory [Bytes]	Linux kernel
Free memory [Bytes]	Linux kernel
Memory used for I/O caches [Bytes]	Linux kernel
CPU usage [percent]	Linux kernel
I/O subsystem average-load-index	Trove-module
I/O subsystem idle-time [percent]	Trove-module
Network average-load-index	BMI-module

Table 1.1: Provider of the introduced performance monitor statistics [3]

Limitations: Due to the trace format data had to be handled multiple times via several intermediate tools for post processing. Furthermore, the statistics for performance counters are not sufficient for the analysis. Therefore, a new tracing environment got developed. It includes HDTrace, a XML based tracing format and the visualization tool Sunshot, a major rewrite of Jumpshot to deal with the new format, are made to support the new PVFS. Moreover, PIOViz provides many additional advantages. They will be discussed later in chapter 4.

1.3 Goal of the Thesis

The goals of this thesis are to add new statistics, extra information of internal process states and embed the new API HDTrace into PVFS2. The number of pending operations of internal layers should be traced. Thus, extended tracing of IO calls within PVFS should be possible. Also, the new tracing environment should be easier extensible.

1.4 Structure of the Thesis

Chapter 2 describes the architecture of PVFS, MPICH2 and the main project PIOViz. In chapter 3 the internal interactions and relations of the PVFS layers are described. Chapter 4 provides design considerations of the project. Chapter 5 describes some non-trivial modifications of the implementation. Chapter 6 evaluates the approach, experiments on the working group's cluster are conducted and shows the effectiveness of the new PIOViz environment. Chapter 7 summarizes and concludes of this project.

[3] These statistics are taken from [Kun07] with the author's permission

2 System Overview

This section gives a general overview of the software environment relevant for this thesis including PVFS server and client components, the MPI-2 implementation MPICH2 and PIOViz, a visualization environment which allows tracing client and server activities. [1]

2.1 The Parallel Virtual File System PVFS

Nowadays there are many scientific computations using very large computer clusters due to the demand for high performance. These clusters have many disks located in different nodes and managed by a software which is called *distributed file system*. In addition to a distributed file system a *parallel file systems* allows concurrent access to distributed storage by splitting single files into multiple *subfiles* located on distinct nodes.

The Parallel Virtual File System PVFS is one open source parallel file system which provides high performance for parallel applications, where concurrent accesses to large I/O and many files are common. PVFS provides distribution of I/O and metadata, avoiding single points of contention, and should allow to scale high-end terascale and petascale systems [Proa].

In the context of the file system the following terms are important:

PVFS server: In PVFS (version 1) there were two types of servers: data and metadata servers. Data servers store data in a round robin manner, typically striped over multiple nodes. Each server uses a local file system to store data. Metadata servers store object **attributes**. This is all the information about files in the UNIX sense, i.e. object type, ownership, permissions, timestamps, and filesize. Additional information like extended attributes and the directory hierarchy is stored on metadata servers, too. In PVFS2 there is only one type of server, the *pvfs2-server*. Each *pvfs2-server* runs normally on a different node and can act as data or metadata server according to an external configuration file. If desired, a node can have more than one pvfs-server instance.

PVFS client: In the following the term *client* is used for processes (or nodes) that access the virtual file systems provided by the PVFS servers. Applications can use one of the available userlevel-interfaces to interact with the file system.

File system objects: Objects which can be stored in a PVFS file system are files, directories and symbolic links. Internally PVFS knows additional system level objects: metafiles, which contain metadata for a file system object, datafiles [2], which contain a part of a logical file's data and directory data objects, which store the mapping of a filename to a handle. PVFS stores a file system object as one or

[1] This chapter is based on [Kun07] with the author's permission

[2] In the context of other parallel file systems often the term subfile is used as a synonym

multiple system level objects.
As a concrete example a logical file is stored as a metafile and the file data is split into one or more datafiles, which can be distributed over multiple or even all available data servers. The metafile containing the object's attributes and other metadata for a single PVFS file system object is located on exactly one of the available metadata servers.

Handle: A handle is a number identifying a specific internal object of a PVFS filesystem and is similar to the ext2 inode number. The mapping between a handle to the servers responsible for this particular object is specified in an external configuration file.

2.1.1 Software Architecture

PVFS uses the layer model illustrated in figure 2.1. Interfaces for the layers use a non-blocking semantics. Desired operations are first posted and then their completion status is tested. A unique identifier is created for every post, which is used as an input for test calls. Also there are test calls which test on multiple or all operations of a context. In case the operation is not very complex and time consuming, it is possible that the operation completes immediately during the post call. This has to be checked by the caller. For a more detailed description refer to [Tea03].

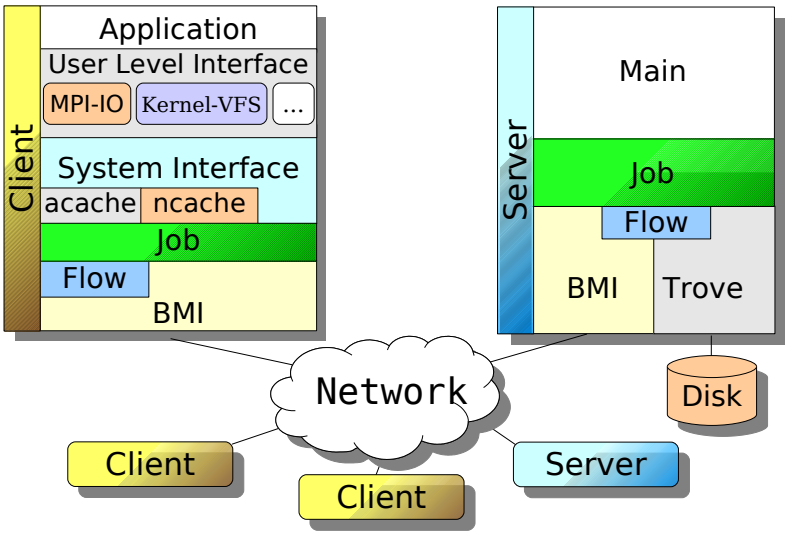

Figure 2.1: PVFS software architecture

Userlevel-interface

The userlevel-interface provides a higher abstraction to the PVFS file system. There are currently two userlevel-interfaces available: a kernel interface and an MPI[3] interface. The kernel interface is realized by a kernel-module integrating PVFS into the kernel's Virtual File System Switch (VFS) and a user-space daemon which communicates with the servers. PVFS is especially designed to provide an efficient integration into any implementation of the Message-Passing Interface specification MPI-2.

[3]Message Passing Interface

System interface

The system interface API provides functions for the direct manipulation of file system objects and hides internal details from the user. Applications can use the **libpvfs** functions to access the PVFS file systems. A program using this interface is considered as client. Invoking a system interface function starts a statemachine which processes the operation in small steps.

Client-side caches: Several caches are part of the system interface and try to minimize the number of requests to server processes. The attribute cache (acache) manages metadata like timestamps and handle number.
The name cache (ncache) stores a file system object's filename and related handle number. To prevent the caches from storing invalid information, data is valid only within a defined timeframe [4] and gets invalidated when the server signals the client that the object it tries to operate, does not exist.

Job

The job layer consolidates the lower layers BMI, Flow, and Trove into one interface. It also maintains threads and callback functions, which will be given as input to called functions. On completion of an operation the lower layers can simply run the callback function, which knows the next working step necessary to finish the request. This can be used in the persistency layer, for example, to initiate a transmission when data was read. Furthermore, a request scheduler, which decides when incoming requests get scheduled, can be considered as part of the Job layer. Main function of the scheduler is to prevent conflicting operations on metadata.

Flow

A flow is a data stream between two endpoints. An endpoint is one of memory, BMI, or Trove. The user can choose between different flow protocols defining the behavior of the data transmission. For example, buffer strategy and number of messages transferred in parallel may be different for two flow protocols. To initiate a data transfer, flow has to know the data definition (size, position in memory, ...) and the endpoints. Flow then takes care of the data transmission. Complex memory and file datatypes are automatically converted to a simpler data format, convenient for the lower level I/O interfaces.

BMI

The Buffered Message Interface provides a network independent interface for message exchange between two nodes. Clients communicate with the servers by using the request protocol, which defines the layout of the messages for every file system operation. BMI can use different communication methods, currently TCP, Myricom's GM and Infiniband are included in the source package. Similar to MPI, BMI requires to announce the receiving of a message before the message is expected to arrive. An announcement includes the sender of the message, expected size of the message and identification tag.

[4] Timeout corresponds to the storage hint **HandleRecycleTimeoutSecs**

Sometimes it is not possible to know the origin of the message. Then, this message is called unexpected message. A client starts a file system operation by sending an operation specific request message to the server. However, the server cannot know that there will be a request or might not be aware of the client's existence. So the server buffers a pool of unexpected messages, which have the maximum possible size of an initial request.

Trove

Trove provides and administrates the persistent storage space for system level objects. Data is either stored as a keyword/value (keyval) pair or as a bytestream. Keyword value pairs are used to store arbitrary metadata information while byte streams store a logical file's data. Bytestream data is accessed in contiguous blocks using a size and an offset, while keyval data can be accessed by resolving the key. Also for each internal storage object there is a set of common attributes stored.

Like BMI and Flow, Trove can switch between different modules, which are actually different implementations of the whole interface. For more information refer to section 3.3.2.

Server main loop

The server main loop checks the completion of the statemachines processed by threads. If a statemachine's current state is finished, the next state is assigned for work and a particular state function is called. This either completes the current state's operation or enqueues the operation for the threads. In case the function completes immediately, the next state of the statemachine is called directly. There are BMI, Flow and Trove threads, which take care of the unfinished operations depending on the operations' type.

In addition, unexpected messages from BMI are decoded. For each message the appropriate statemachine is started and a buffer for a new unexpected message is provided.

Performance Monitor

Each server has a central performance monitor (internally also referred as performance counter) orthogonal to the layered architecture, which allows a key based access to a set of 64 bit long statistics. These statistics are maintained and stored for intervals of a configurable length (default: 1000 ms, configuration option: **PerfUpdateInterval**). A history keeps the statistics a number of intervals (default: 5 intervals). Operations offered by the performance monitor interface are addition, subtraction or setting of a value specified by a key. Flow for example uses this interface to store the number of bytes read within the interval.

2.2 MPICH-2

MPICH-2 is a high-performance and widely portable implementation of the Message Passing Interface (both MPI-1 and MPI-2). The goals of MPICH2 are:

- to provide an MPI implementation that efficiently supports different computation and communication platforms including commodity clusters (desktop systems, shared-memory systems, multicore architectures), high-speed networks (10 Gigabit Ethernet, Infiniband, Myrinet, Quadrics) and proprietary high-end computing systems (Blue Gene, Cray, SiCotex).
- to enable cutting-edge research in MPI through an easy-to-extend modular framework for other derived implementations [Paga].

A part of MPI-2 is the definition of an I/O interface, which is often referred as MPI-I/O. ROMIO is a portable implementation of this interface. It is built on top of an abstract-device interface for sequential and parallel I/O (ADIO [TGL96]). ADIO abstracts from the actual used file system and supports many file systems like POSIX conform file systems, NFS, or inherently parallel file systems like PVFS. MPI-I/O calls are directly processed by ROMIO, which invokes the appropriate ADIO methods.

An excerpt of independent software components and the relevant software stacks are shown in figure 2.2. MPE is described in the next section.

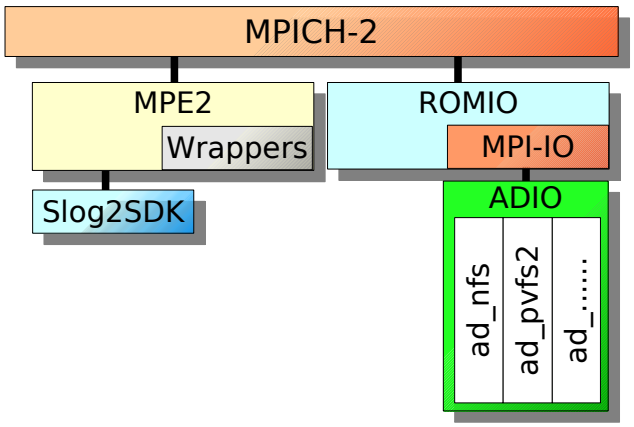

Figure 2.2: MPICH-2 software components

There are two optimization mechanisms incorporated in ADIO, data-sieving and collective I/O [TGL99].

Data-sieving is used when a process independently accesses a non-contiguous piece of data in a file. If the holes between the accessed data regions and the data fits into a buffer with a user specific size (default: 4 MByte for read and 512 KByte for writes) only this contiguous block is fetched. For writes read-modify-write phases are necessary. This requires to first read the old data which should be preserved by PVFS, then modifications of the buffered data are made and at last the data is written back in a single contiguous block. Consequently, file locking is necessary to ensure consistency, which not yet supported.

Collective I/O is optimized by a two-phase-protocol. First the requested data portions are broadcasted to all processes and analyzed. The processes decide if the accesses should be merged, if not each process does individual I/O operations with data-sieving. In case I/O should be done collectively, file regions are assigned to a set of aggregator nodes (normally all available clients) in a granularity of the collective buffer size (default: 4 MByte). For collective reads, the aggregators then repeat the following

two phases with data portions fitting in their collective buffer. In the first phase the aggregators read their current file portion, and then in phase two the data is distributed to the clients which requested the data. Communication phase and I/O phase are switched for write operations. These optimizations are described in [TGL99].

Summary: This chapter described the software environment consisting of MPICH-2 with MPE and ROMIO, and the parallel file system PVFS2. Furthermore, the software architecture and the tasks of the different layers in PVFS2 are outlined.

3 Internal View of PVFS

This chapter highlights some internal details of PVFS components, which are important to understand the PVFS structure and the implementation [1].

3.1 Client and Server Interaction

This section highlights various aspects of internal request processing. Internal processes, which are triggered by a system interface call on the client and server are shown. The new PVFS hint, used to exchange process information, is also introduced. In addition, the basic concept of statemachines, and both client and server statemachines used in PVFS are described. Moreover, we will discuss the request scheduler, which manages request jobs.

3.1.1 System Interface Call

In order to export a new request to the user level interface one has to add new system interface calls to the system interface. The prototypes have to be put either in the file **include/pvfs-sysint.h** or for management function in **include/pvfs-mgmt.h**. Management functions typically include more knowledge about the parallel file system interns, for example, they are useful for file system checking tool and thus not intended to be used by regular users directly.

Most available system interface calls have non-blocking versions available to allow concurrent execution in the application. As a natural way to get both variants the blocking versions call internally the asynchronous version followed by a **PVFS_wait** call. Similar to the **MPI_wait** call this function drives the asynchronous operations and blocks until the specified operation completes. In order to connect both calls the asynchronous operation returns an *operation id* which is a unique identifier on the client. The asynchronous system call also includes a so called user pointer, which may be filled with arbitrary data associated with the particular call in order to allow a simple mapping between operation and operation ID without storing the mapping explicitly. The prototypes for the file creation call are shown with extended comments in listing 3.1.

Listing 3.1: Create system calls

```
PVFS_error PVFS_isys_create(
    /* logical objects name */
    const char *entry_name,
    /* parent directory reference */
    PVFS_object_ref ref,
```

[1] This chapter is partly based on [Kun07] with the author's permission

```
    /* attributes which should be set on the new object */
    PVFS_sys_attr attr,
    /* identification of system interface user */
    const PVFS_credentials *credentials,
    /* distribution of the new file */
    PVFS_sys_dist *dist,
    /* pointer to the output of the function call */
    PVFS_sysresp_create *resp,
    /* operation id, necessary for wait */
    PVFS_sys_op_id *op_id,
    /* hint, necessary for process interaction*/
    PVFS_hint hints,
    /* can be filled with arbitrary data from the caller */
    void *user_ptr);

PVFS_error PVFS_sys_create( /* calls the non-blocking function */
    char *entry_name,
    PVFS_object_ref ref,
    PVFS_sys_attr attr,
    PVFS_credentials *credentials,
    PVFS_sys_dist *dist,
    PVFS_sysresp_create *resp
    PVFS_hint hints);
```

3.1.2 PVFS Hint

In PVFS, hints are used for saving additional attributes or information. This information can be transported between client and server or just being used internally. Hints use the key/value concept to store data. Hints are encoded and added to a hint list, e.g. via the function **PVFS_hint_add**. Similarly, hints can be decoded and taken from the hint list, e.g. via the function **PINT_hint_get_value_by_name**. At the system interface hints can be committed to most system calls, and new hints can be added internally. The prototypes of PVFS hints are shown in listing 3.2.

Listing 3.2: Prototypes and internal structure of PVFS hints

```
typedef struct PVFS_hint_s
{
...
    char *type_string;          /* name of hint */
    char *value;                /* value of hint */
    int32_t length;             /* length of hint */
    /* function interfaces for encoding/decoding hint */
    void (*encode)(char **pptr, void *value);
    void (*decode)(char **pptr, void *value);
...
    struct PVFS_hint_s *next; /* next hint of the hint list */
} PINT_hint;
```

```
int PVFS_hint_add(
    PVFS_hint *hint,
    const char *type,
    int length,
    void *value)

void * PINT_hint_get_value_by_name(
    struct PVFS_hint_s *hint,
    const char *name,
    int *length);
```

3.1.3 Statemachines

In PVFS a statemachine consists of a unique name, multiple states, each with a state name and transitions between them. Basically, in each state a statemachine can do one of the following operations:

- Invoke a specified funtion.
- Invoke another statemachine.
- Invoke the default transition.

The next state transition is determinated by the return value of this function or nested statemachine. Besides, state functions require two qualities: they should need little time and must not influence the results of different statemachines running concurrently. Therefore, it is allowed to start one possible blocking (BMI or Trove) operation in a non-blocking fashion per state. Using such a model allows simulating complex operations possible, which means statemachines can be nested to model and simplify the handling of common subprocesses.

Integration of new statemachines: The specification of a statemachine is easy and done in a separate statemachine file (**.sm**). State machine files differ from normal C code by an additional section for the statemachine, which begins and ends with two per cent (%) signs. This section is transformed by the parser **src/common/statecomp** into the corresponding C source code during the build process.

The example 3.3 shows an example of the internal statemachine structure.

Listing 3.3: An example of PVFS state machine definition

```
%%
machine example_sm
{
    state init
    {
        run func_init;
        success => next_state;
        default => cleanup;
    }

    state next_state
```

```
    {
        jump next_sm;
        success => another_next_state;
        default => cleanup;
    }
...
    state cleanup
    {
        run func_cleanup;
        default => terminate;
    }
}
%%
```

A statemachine's name is given after the keyword **machine**. A set of states follows, each introduced by **state**. The first line of each state indicates which function or nested statemachine is executed. While **run** indicates a function is executed, **jump** indicates a nested statemachine. Each function or nested statemachine has a return value to determine the next state. In case the function is terminated successfully, the next state is called, otherwise the **default** state is called. A statemachine begins with the **init** state and ends with the **terminate** state [Kuh07].

3.1.4 Client Statemachine

The asynchronous system interface call instantiates a new statemachine with the type of the call on the client and posts it. Depending on the call requests are sent to one or multiple servers during the processing of the statemachine. In order to exchange information between the states (otherwise they are stateless) needed data of each statemachine is stored in the union *u* of the structure *PINT_client_sm*[2]. See listing 3.4 for an excerpt of the structures. The system interface calls use this union to put the input parameters in the struct of the appropriate function call.

Listing 3.4: Client statemachine structures

```
struct PINT_client_create_sm
{
    char *object_name;                  /* input parameter */
    PVFS_object_attr attr;              /* input parameter */
    PVFS_sysresp_create *create_resp; /* in/out parameter */

    int retry_count; /* number of create retry attempts */
    int num_data_files; /* number of datafiles of the new file */
    /* number of requested datafiles of user */
    int user_requested_num_data_files;
    int stored_error_code;

    PINT_dist *dist; /* distribution of new logical file */
    PVFS_sys_layout layout;
```

[2]The structure is located in the file **src/client/sysint/client-state-machine.h**

```
    PVFS_handle metafile_handle;
    int datafile_count;
    PVFS_handle *datafile_handles; /* array of created datafiles */
    int stuffed;
...
};

typedef struct PINT_client_sm
{
...
    /* used internally by client-state-machine.c */
    PVFS_sys_op_id sys_op_id;
    void *user_ptr;

    /* user and group */
    PVFS_credentials *cred_p;

    PVFS_hint hints;
    union
    {
        struct PINT_client_remove_sm remove;
        struct PINT_client_create_sm create;
...
    } u;
} PINT_client_sm;
```

3.1.5 Server Statemachine

The server waits for incoming messages. New requests are first decoded and then depending on the operation type, a new statemachine is started. The mapping between operation type and statemachine is done in the file **src/server/pvfs2-server-req.c**. In addition a name for better debugging is specified. Furthermore, required attributes of the object working on is indicated and if the permissions should be checked.

Listing 3.5: PVFS server request mapping table

```
/* table of incoming request types and associated parameters */
struct PINT_server_req_entry PINT_server_req_table[] =
{
       /* 0 */ {PVFS_SERV_INVALID, NULL},
       /* 1 */ {PVFS_SERV_CREATE, &pvfs2_create_params},
       /* 2 */ {PVFS_SERV_REMOVE, &pvfs2_remove_params},
       /* 3 */ {PVFS_SERV_IO, &pvfs2_io_params},
...
       /* 16 */ {PVFS_SERV_MGMT_SETPARAM, &pvfs2_setparam_params},
...
}
```

In order to make the statemachine available the name has to be put into the header file **src/server/pvfs2-server.h** like for the create statemachine:

```
extern struct PINT_state_machine_s pvfs2_create_sm;
```

3.2 Request Scheduler

Beside the components already described in section 2.1, PVFS offers a request scheduler on the server side for management of request jobs. This scheduler uses a hash table, which contains a linked list for saving requests. Moreover, it provides two function interfaces for adding new requests and searching a specified request.

```
struct qhash_table
{
    struct qhash_head *array;
    int table_size;
    int (*compare) (void *key, struct qhash_head * link);
    int (*hash) (void *key, int table_size);
...
};
```

Internally, each request has many attributes for execution and a request state which tells the server if the request can be performed directly, later, at a particular time, or if it is already finished.

```
struct req_sched_element
{
    enum PVFS_server_op op;             /* server operation type */
    struct qlist_head list_link;        /* ties it to a queue */
    struct qlist_head ready_link;       /* ties to ready queue */
    void *user_ptr;                     /* user pointer */
    req_sched_id id;                    /* unique identifier */
    struct req_sched_list *list_head; /* points to head of queue */
    enum req_sched_states state;        /* state of this element */
    PVFS_handle handle;
    struct timeval tv;                  /* used for timer events */
    /* indicates type of access needed by this op */
    enum PINT_server_req_access_type access_type;
};

enum req_sched_states
{
    /* request is queued up, cannot be processed yet */
    REQ_QUEUED,
    /* request is being processed */
    REQ_SCHEDULED,
    /* request could be processed, but caller has not
    asked for it yet */
```

```
    REQ_READY_TO_SCHEDULE,
    /* for timer events */
    REQ_TIMING,
};
```

3.3 Layer Internal Processing

This section provides more detailed information about the internal I/O layers.

3.3.1 Post and Test of Operations

As discussed in section 2.1.1, the model for internal I/O is that a desired operation is first posted, then it is tested until the operation has completed, and finally the return value of the operation is checked to determine if it was successful. Every post results in the creation of a unique ID, that is used as an input to the test call. Currently, there are three possible classes of return values for posting operations:

- 1 indicates that the operation is already completed.
- 0 indicates that the post was successful, but has not completed yet and still runs in the background. The caller needs to test for its completion.
- Another value indicates that the post failed due to an error.

PVFS provides three test fuctions to check the completion states of operations. Each internal layer supports the following variants of the test function (where PREFIX depends on the API)[Prob]:

- **PREFIX_test()**: checks for completion of an individual operation based on the ID given by the caller.
- **PREFIX_testsome()**: This is an expansion of the above test function. The difference is that it uses an array of IDs and a number of elements as input, and provides an array of status values and a number of completed operations as output. Any non-zero ID in the array will be checked for completion. With this mechanism one can improve efficiency by checking for completion of many operations.
- **PREFIX_testcontext()**: This function is similar to PREFIX_testsome(). Instead of using an array of IDs as input, it tests for completion of any operations that have previously been posted, regardless of the ID. These operations belong to a context, which must be created and then posted to a particular interface. This context is then used as an input argument to every subsequent post and test call. It is useful to ensure that it does not return information about operations, that were posted by different callers.

3.3.2 Trove

This section provides more information about I/O methods and types of I/O requests in Trove.

I/O Methods

In PVFS there are four methods of I/O which specify how data and metadata are stored and managed by PVFS servers.

- **alt-aio**: This is the default method. It uses a thread-based implementation of asynchronous I/O. It is recommended for local storages, including RAID setups. In this method each I/O operation spawns a thread. The maximum number of concurrent threads is limited. Further operations get queued.
- **directio**: It uses a direct I/O implementation to perform I/O operations to datafiles. This method may lead to significant performance improvement if PVFS servers are running over shared storages, especially for large I/O accesses.
- **null-io**: This method is an implementation that does no disk I/O at all and is only useful for development or debugging purposes. It can be used to test the performance of the network without doing I/O to disks.
- **dbpf**: This method uses the system's Linux AIO implementation. It is not recommended in production environments any more.

The I/O method can be selected in the PVFS server config file. An excerpt is shown in listing 3.6.

Listing 3.6: Excerpt from a pvfs config file

```
<Filesystem>
    Name pvfs2-fs
...
    <StorageHints>
        TroveSyncMeta yes
        TroveSyncData no
        TroveMethod alt-aio
    </StorageHints>
</Filesystem>
```

I/O Requests

There are two types of I/O requests defined by the request protocol (see listing 3.7):

I/O: This is the normal request I/O used for large amounts of data. In the data structure of PVFS this type will be handled via Flow. It requires to first announce I/O, then data is transferred.

Small I/O: This is used for very small access which server can process directly without Flow. In contrast to the I/O request, the overhead is reduced.

Listing 3.7: I/O types

```
typedef struct PINT_server_op
{
...
    union
    {
        ...
        struct PINT_server_io_op io;
        struct PINT_server_small_io_op small_io;
    } u;
...
} PINT_server_op;

/* small I/O */
struct PINT_server_io_op
{
    /* contain I/O information of Flow */
    flow_descriptor* flow_d;
};

/* small I/O */
struct PVFS_servresp_small_io
{
    enum PVFS_io_type io_type;

    /* the io state machine needs the total bstream size to
     * calculate the correct return size */
    PVFS_size bstream_size;

    /* for writes, this is the amount written.
     * for reads, this is the number of bytes read */
    PVFS_size result_size;
    char * buffer;
};
```

Note that PVFS provides support for non-contiguous I/O requests, i.e. it is possible to access multiple regions in a file with one request.

Summary: Some internal structures of PVFS client, server and Trove have been discussed to show interaction between these components. Statemachines are a key concept in PVFS and used for management of requests. Furthermore, hints, a new data structure of PVFS, were also introduced. Hints are used for storing meta information between processes.

Next, new metrics and information used for tracing will be discussed.

4 Design

In this chapter considerations are presented, that arose during the design phase of the project. At first, information and metrics are assessed which allow users to analyze the activities within the system. Furthermore, alternatives for the realization of these metrics and the overall tracing architecture are discussed.

4.1 Overview

To tune a complex system it is important to know which components are related to bottlenecks. There are many reasons for bottlenecks, the most important of which can be listed as follows:

- **Hardware**: Normally, client and server nodes in the cluster have different types of hardware, including RAM, CPU, network, disk and especially cache which is used for I/O operations. Hardware difference might lead to system unbalanced. For example, a node with 2 GHz CPU works faster than a node with 1 GHz CPU. Besides on one particular node the hardware can be intermediately broken, or used heavily in comparison to other nodes because of any background processes which stress it a lot. In the parallel system environment in which every node is a part of parallel jobs, this node can lead to bottlenecks. As the consequence, tracing hardware is important for the analysis.
- **Software**: Usually, in the cluster different file systems can be found as well as multiple MPI client programs. This can lead to bottlenecks due to the different I/O demand. Moreover, bottlenecks can even appear in PVFS which manages the distribution of I/O operations. There are many internal layers within PVFS, and we have to find out what information or metrics are useful for the analysis. In this thesis included information is as follows:
  - *Number of concurrent operations*: This metric is defined as the number of pending operations running in a particular layer of PVFS. It can be useful for comparison of load between different nodes in the cluster.
  - *Statemachine*: This is the fundamental concept in PVFS as discussed in section 3.1.3. Therefore, it is important to understand the processing sequences of statemachines. Based on this information we can analyze why some operations were slow or behaved abnormally. In addition, tracing related to I/O are also needed and described later.
  - *Correlations*: Knowing how layers in PVFS work is good, but rather not enough. We also need to understand their connections and interactions, for example, how operations are processed, which client operations call which server operations, etc.

4.2 General Tracing Considerations

Important considerations regarding tracing can be listed as follows:

Code integration: We might ask ourselves whether the tracing API should be embedded into PVFS or run independently. One thing we could keep in mind is that we can modify PVFS but must not change its semantics. Also, code modifications should be minimized to reduce porting effort of PVFS updates. We have decided to embed the tracing API into PVFS because it is easier to implement, and in addition we can use some internal metrics of PVFS to enhance tracing aspects. To reduce porting complexity, some files will be added.

Controlling tracing: Tracing causes overhead, so we have to think how to reduce or minimize it. Users should be able to control the tracing on servers remotely via a tool. Servers might run 24/7 and it is not acceptable to define tracing behavior only during startup, or to create traces only during shutdown of the server processes. There should be at least three parameters for this tool, one for the tracing activation, one for deactivation, and another one for choosing events which are traced. We use the program **pvfs2-set-eventmask** [1] to generate a management request to control server tracing. For more information refer to section 5.1.3. After deactivation trace data is saved in trace files and can be visualized via Sunshot.

Tracing accuracy:
Periodic update: In PIOViz 1.0 events were traced and updated in a fixed time period. Hence, not all information in trace files was needed (in idle periods), or detailed enough (in active phases). If requests change rapidly the results are inaccurate.
On demand: Events are traced on demand, which means information is only traced if the situation changes. However, if traced information changes rapidly the overhead increases compared to periodic updates, information is accurate. See section 4.3.2 for more information.

Thread safety: Another thing to consider while tracing in PVFS is thread safety. In such an environment like PVFS where parallel accesses are permitted, one must check for thread-safety, otherwise the results might be wrong if they are not handled correctly.

4.3 Tracing Metrics and Information

This section provides more detailed information about factors of bottlenecks which are mentioned in section 4.1.

4.3.1 Hardware Statistics

PIOViz 1.0 provides some hardware statistics which are already explained in section 1.2.2. They are mostly taken from the kernel to let us know what is going on in the system. HDTrace provides Resources Utilization Tracing Library (RUTL), a performance trace library which collects statistics

[1] The source file is located in src/apps/admin/pvfs2-set-eventmask.c

from the */proc* interface in a periodic time by utilizing libGtop. These statistics will be saved in the new format of HDTrace. In detail, PTL provides the following statistics [2]:

- aggregated load of all CPUs.
- CPU load for each single CPU.
- amount of main memory used.
- amount of free main memory.
- amount of shared main memory.
- amount of main memory used as buffer.
- amount of main memory cached.
- incoming traffic of each network interface.
- outgoing traffic of each network interface.
- aggregated incoming traffic of external network interfaces.
- aggregated outgoing traffic of external network interfaces.
- aggregated incoming traffic of all network interfaces.
- aggregated outgoing traffic of all network interfaces.
- amount of data read from hard disk drives.
- amount of data written to hard disk drives.

Compared to the PIOViz 1.0 approach, the new approach supports better portability and extendability. Also, the same statistics are available in clients as well.

4.3.2 Number of Concurrent Operations

This metric is defined for BMI, Flow, Trove, requests (number of statemachines) and blocked requests. For each of these the number of operations is traced accurately, i.e. on modification the value is updated. Requests and blocked requests differ from their states which are managed by the request scheduler (see section 3.2). For instance, if there are multiple metadata requests accessing the same directory, only one of them is permitted to access the file while the others are blocked by the request scheduler to protect data consistency. In general, we need two functions for counting (increase/decrease) the number of concurrent operations. These functions use the tracing API of HDTrace to save the current counter results in trace files. Basically, they are placed around operations of PVFS layers and information mentioned above (see listing 4.1). Via this metric we know how many operations actually run in the meantime and how long they were processed, etc.

Listing 4.1: An pseudo-code example for counting the number of concurrent operations

```
/* X can be BMI, Flow, Trove, requests and blocked requests */
increase_counter_for_X
operation_of_X
decrease_counter_for_X
```

Note that the nonblocking semantics of the internal layer require to check multiple possible states.

[2] The source file is ResourcesUtilizationTracingLibrary/include/RUT.h

4.3.3 Statemachine Tracing

This section describes the structure of a client/server statemachine in PVFS.

Statemachine Processing

In PVFS for each client request there is a corresponding client statemachine started. One goal is to trace statemachine transitions. In principle, two functions of the tracing API are used. Similar to section 4.3.2, they are placed around needed functions in states. One function writes names and needed attributes of states while the other closes the transition. For more information refer to section 4.3.3. In brief, a statemachine begins with *PINT_state_machine_start*, runs with *PINT_state_machine_invoke*, *PINT_state_machine_continue* and *PINT_state_machine_next*, and ends with *PINT_state_machine_terminate*. Besides it is interesting that both client and server use the same statemachine structure. Therefore, we can use this advantage to reduce tracing codes. The states during lifetime of a statemachine are shown in figure 4.1 and described as below:

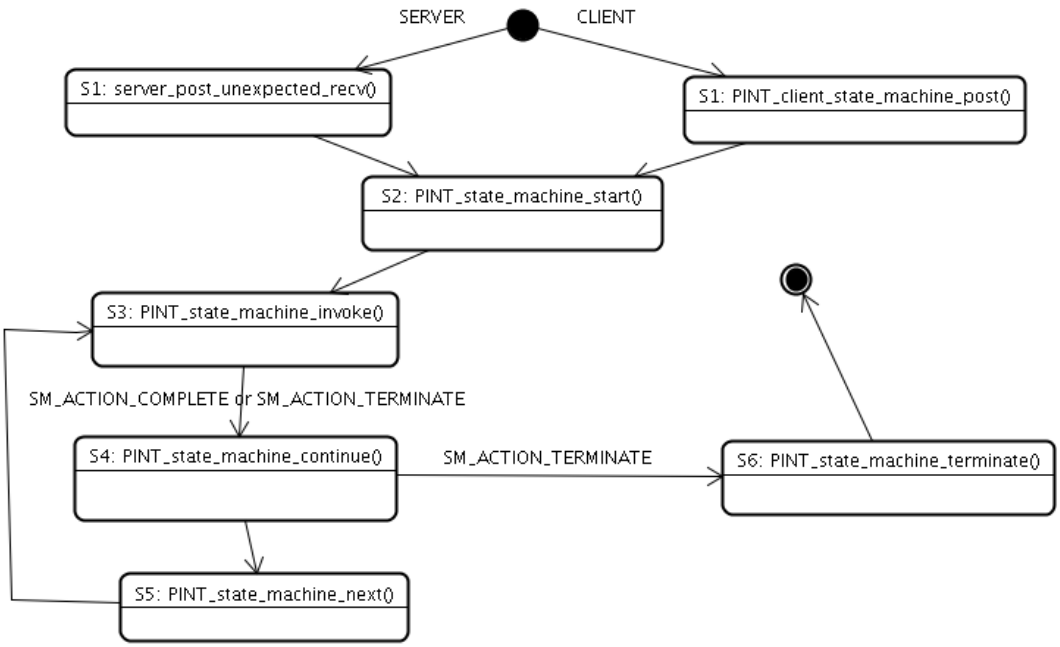

Figure 4.1: State chart of processing of a statemachine

S1 Client or server allocate space for a statemachine, and invoke code to the statemachine stack which manages the statemachines.

S2 On the top of the stack the current (nested) statemachine is started.

S3 While the statemachine runs, the state *PINT_state_machine_invoke* tries to invoke the state's action function. There are four return values of a state action function: *SM_ACTION_DEFERRED*, *SM_ACTION_COMPLETE*, *SM_ACTION_TERMINATE* and *SM_ACTION_ERROR*, each is associated with a corresponding operation. In case of *SM_ACTION_DEFERRED*, the statemachine will be registered back in the statemachine stack, so that it can be handled later by client. In case of *SM_ACTION_ERROR*, the client statemachine will log needed information and then release resources associated with the statemachine.

S4 The next state *PINT_state_machine_continue* is reached which means the return value is either *SM_ACTION_COMPLETE* or *SM_ACTION_TERMINATE*. In case of *SM_ACTION_TERMINATE* the state *PINT_state_machine_terminate* is reached, otherwise *PINT_state_machine_next*.

S5 The state *PINT_state_machine_next* manages a stack of nested states. The first state of the stack will be removed and handled by *PINT_state_machine_invoke* (like in S3). This procedure repeats in a loop manner, and is only finished if the stack is empty or the return value of any state action function is set to *SM_ACTION_TERMINATE*. Then the state *PINT_state_machine_terminate* is reached.

S6 The state *PINT_state_machine_terminate* cleans up the environment, and terminates the statemachine.

Modifications for Tracing States

In principle, there are two kind of functions for tracing states of HDTraceWritingLibrary [3]. Let us assume, they are called *START* and *STOP*. Basically, the *START* traces events of a transition, and the *STOP* closes the transition.

In state *PINT_state_machine_start* we use *START* to trace the statemachine's name before *PINT_state_machine_invoke* is run. In *PINT_state_machine_invoke* we place *START* and *STOP* round the state action function to trace the state name of this function. Similarly, in *PINT_state_machine_next* they are also placed around each nested state of the nested state stack to trace its state name. In *PINT_state_machine_terminate STOP* is placed to close the statemachine transition.

State Overlapping

It can happen that two or more statemachines are interleaved because states can do work concurrently. If states return with *SM_ACTION_DEFERRED*, in this case, it is impossible to know how many concurrent operations will be performed, maybe multiple per client.

Fortunately, Sunshot has solved this problem by allowing us to expand the process time line from one to multiple. In case of the state overlapping problem, we will see a sign X, and can expand the time line to see the nested states, or statemachines inside. In other words, we don't need to think about other solutions to overcome this problem. The solution is shown in figure 4.2.

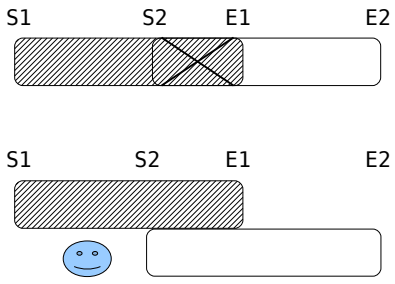

Figure 4.2: Multiple time lines for state overlapping

[3] The source file is in HDTraceWritingLibrary/include/hdRelation.h

4.3.4 Correlation Creation

The correlations in PVFS are introduced in section 4.1. In this thesis apart from the correlation between client and server, we will create the correlation between request and the layer Trove due to I/O issues. The correlation between server and other PVFS layers could be done as future work.

First we will discuss the data structure used in the tracing API before going into the details on how to create correlations. In HDTrace each process is associated with an unique ID. To create connections between two processes we have to relate their IDs by creating a 'relation' containing the process IDs in a particular format. This relation is transported via hints (see section 3.1.2). Because there are two types of correlations, two new types of keys are added; one for client/server (*c_key*), and another one for server/trove (*s_key*).

The details are shown in figure 4.3, and described as below:

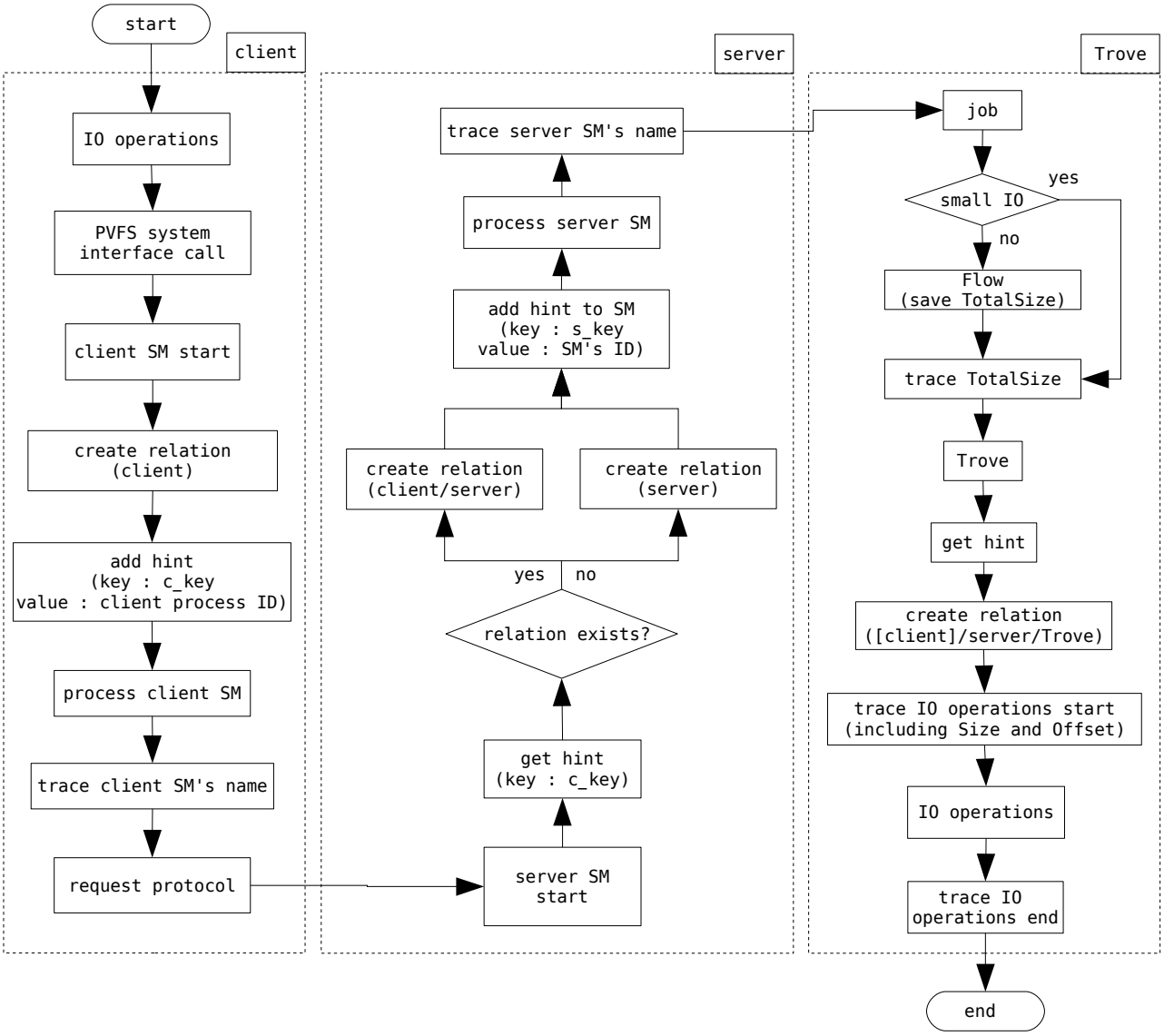

Figure 4.3: Create correlations in PVFS

Client side:

- At the beginning, I/O applications use system interface calls to access the file system.
- As described in section 4.3.3, for each client request there is a corresponding client statemachine started in PVFS. The statemachine creates a relation containing the client process ID.
- The relation is saved in a hint with *c_key*. The hint is then added to the hint list of each client request.
- The client statemachine uses the client request to trace statemachine events and sends the hint list to the server via the request protocol. After the client statemachine is finished, the hint list is removed from the memory.

Server:

- The server checks unexpected messages for new requests. If any exist, corresponding server statemachines will be started to handle them. Let us assume we have one server statemachine started.
- In the hint list from the client request the server statemachine searches for the relation via *c_key*.
- If this hint exists, the server statemachine creates another relation containing the client ID via the old relation and the ID of this server statemachine request. Otherwise it creates a relation for this request.
- If the relation is saved in a hint with *s_key*, then the hint is added to the hint list of the server statemachine.
- The hint list is transported to Trove.

Trove:

- In Trove a corresponding Trove non-blocking operation is started which searches for the relation via *s_key* in the hint list.
- If the relation exists, Trove creates another relation containing the request ID from the old relation and the Trove ID request. Now we can identify the parent statemachine and implicit the calling client as well.
- I/O operations are performed and traced at the same time.

Tracing I/O Details

For the analysis of I/O the following information is useful:

- *Total Size*: The total amount of data which will be transferred. It needs to be traced only once for a normal I/O request (in Flow) or small I/O (see section 3.3.2).
- *Size*: The amount of data transferred with each operation. It is traced for each Trove operation.
- *Offset*: File position of each single I/O operation.

4.4 Generating Trace Files and Postprocessing

This section highlights the tracing workflow of in this project. The creation of trace files and their relations are described. Files generated during client and server tracing and postprocessing to generate project files is shown in figure 4.4.

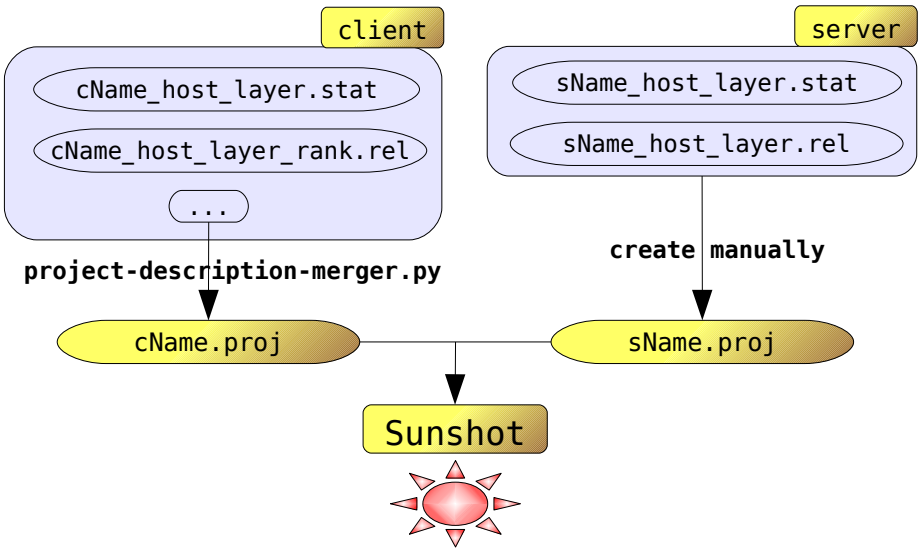

Figure 4.4: Generating trace files and post processing

In principle, tracing client and server is independent from each other and configured by the user. To enable the tracing, users have to specify event names which should be traced. To enable tracing of specific events the program **pvfs2-set-eventmask** [4] with the parameter 'enable' is used (see 'Controlling tracing' in section 4.2). To specify event names of client a particular environment variable is set. So the tracing of client/server is started, and trace files are created. The tracing of a client application is stopped automatically when the process finishes. By contrast, users have to stop the tracing of servers via the parameter 'disable' by help of the program **pvfs2-set-eventmask** manually.

Currently, there are two main types of trace files created by HDTrace: statistic (.stat) and relation files (.rel). While statistic files contain statistic information about PVFS2 layers, relation files contain information about statemachines and process correlations. We use two project files (one for client and one for server), which have the file extension .proj and XML contents, to define what should be viewed in Sunshot. In case of clients there are multiple files parsed by the merger **project-description-merger.py** for creating the client project file.

Server:

- Trace files: Both types of trace files have the same prefix name, except for the file extension. They look like 'sName_host_layer.[stat|rel]', with the file extension .stat for statistics and .rel for states. The prefix parts are described as follows:
  - sName: This name can be set via a parameter of **src/apps/admin/pvfs2-set-eventmask.c** (see section 5.1.3).
  - host: Server host name.

[4] The source file is located in src/apps/admin/pvfs2-set-eventmask.c

  - layer: Name of the PVFS layer.
- Project file: The name looks like ‘sName.proj’. It is mandatory that the prefix ‘sName’ is identical to the first prefix of server trace files. In HDTrace files belonging to one project share the file prefix. The server project file must be generated once manually (see appendix A.4.1). In this thesis we will not look into details on how to create project files (for client and server).

Client:

- Trace files: The statistic files look like ‘cName_host_layer.stat’, similar to server statistic files. In addition, the relation file names contain the rank, i.e. ‘sName_host_layer_rank.rel’. This rank is used for client communications. Note the naming depends on the topology.
- Project file: The name is ‘cName.proj’. Similar to the server project file, the prefix ‘cName’ must be identical to the first prefix of client trace files. The client project file is generated with the merger **project-description-merger.py** after each tracing because the names of client trace files are always different due to the extra ‘rank’. Currently, there is no support for PVFS internal tracing of the merger, instead we have to adjust the project file manually. In the future the merger **project-description-merger.py** should be adjusted to overcome this problem.

Summary: Origins of bottlenecks are important to be understood and it is sensible to define new metrics and tracing information as done in this thesis. They are hardware statistics, the number of concurrent operations, statemachine tracing and the correlations between client, server and Trove. In addition, some design considerations for the implementation are discussed and explained. Moreover, a part of the tracing workflow has been shown, so that users can have a good overview of the project. However, so far the server project file must be created manually. In addition, the client project file must be manually adjusted for PVFS internal tracing as well.

Next, more complex or tricky issues while doing the implementation will be explained.

5 Implementation

The internal structure of PVFS and major design decisions of the extensions have been discussed in chapters 3 and 4. This chapter focuses on some techniques and special design approaches while implementing tracing in PVFS based on the new HDTrace API.

5.1 Integration of HDTrace

This section describes the adaption of some HDTrace packages in the build process of PVFS. Changes are kept apart from PVFS updates by creating new source files to control the tracing with parameters. Furthermore, some techniques used to elaborate the new modifications are described.

5.1.1 Build Process

In order to integrate the new API, first we have to modify the file **configure.in**. Currently, we integrate two HDTrace packages HDTraceWritingCLibrary and RUTL, which are responsible for adding statistics, correlations and performance counters into PVFS. In general, each package needs a corresponding module in **configure.in**, that includes the paths to required libraries and header files. Via parameters (–with-hdtrace and –with-rutl) of **configure.in** the corresponding modules will be enabled in PVFS. Moreover, the output must be adapted to give feedback of integrated modules as well. The package HDTraceWritingCLibrary is needed for RUTL, so we have to make sure that HDTraceWritingCLibrary is included, when using RUTL. An excerpt of **configure.in** is shown in listing 5.1.

Listing 5.1: Modifications of configure.in

```
...
// build HDTraceWritingCLibrary if the parameter --with-hdtrace
   is given
BUILD_HDTRACE=
dnl use the hdtrace library
AC_ARG_WITH(hdtrace,
[  --with-hdtrace=path  Use HDTrace library installed in "path"],
    if test "x$withval" = "xyes" ; then
          AC_MSG_ERROR(--with-hdtrace must be given a pathname)
    else
       CFLAGS="$CFLAGS -I$withval/include
              $(pkg-config --cflags glib-2.0)"
       LDFLAGS="$LDFLAGS -L$withval/lib
              $(pkg-config --cflags glib-2.0)"
       LIBS="$LIBS -lhdStats -lhdTrace -lhdRelation
```

```
            $(pkg-config --libs glib-2.0)"
        AC_DEFINE(HAVE_HDTRACE, 1,
            [Define if HDTrace library is used])
        BUILD_HDTRACE=1

        AC_TRY_LINK([#include "hdTopo.h"], [ ],
        ,
        AC_MSG_ERROR(hdtrace library is not usable))
    fi
)
...
// output configuration state of HDTraceWritingCLibrary
if test "x$BUILD_HDTRACE" = "x" ; then
   AC_MSG_RESULT([PVFS2 configured for hdtrace module   :  no])
else
   AC_MSG_RESULT([PVFS2 configured for hdtrace module   :  yes])
fi

// output of RUTL
if test "x$BUILD_HDTRACERUTL" = "x" ; then
   AC_MSG_RESULT([use RUTL together with hdtrace module  :  no])
else
   AC_MSG_RESULT([use RUTL together with hdtrace module  :  yes])
   /* test if HDTraceWritingCLibrary is already included? */
   if test "x$BUILD_HDTRACE" = "x" ; then
        AC_MSG_ERROR(Error hdtrace module is required for RUTL)
   fi
fi
...
```

5.1.2 New Files

There is one new source file – **pint-event-hd.c** – and two header files – **pint-event-hd.h**, **pint-event-hd-client.h** – integrated into PVFS. With new files the modifications are less dependent on PVFS updates. Files are put into the folder **src/common/misc/** because this folder is used for both client and server processes. The file **pint-event-hd.c** contains all function prototypes; **pint-event-hd.h** is its header file. While these files are used to handle internal client/server processes of PVFS, **pint-event-hd-client.h** is used for external control of the tracing. For a more detailed discussion refer to section 5.1.4.

PVFS uses a tree of folders, which are used for different purposes (e.g. apps, client, server, common, io, kernel and proto). In each folder there is a file, namely **module.mk.in**, referring to source files used for the build mechanism. Therefore, we have to modify this file to include the C file **pint-event-hd.c**. An excerpt of the module file is shown in listing 5.2.

Listing 5.2: Add new files in PVFS

```
DIR := src/common/misc
LIBSRC += $(DIR)/server-config.c \
...
          $(DIR)/pint-event-hd.c \
...
SERVERSRC += $(DIR)/server-config.c \
...
             $(DIR)/pint-event-hd.c \
...
LIBBMISRC += $(DIR)/str-utils.c \
...
             $(DIR)/pint-event-hd.c \
...
```

5.1.3 Controlling Tracing

In PVFS we use the program **src/apps/admin/pvfs2-set-eventmask.c** to trigger (enable/disable) the tracing mechanism. The program delivers two parameters -m and -e. Some additional parameters -d and -l are added for this thesis. They are characterized as follows:

- -m : The mount point of PVFS2.
- -e : Enable tracing of particular events, as a comma separate list for multiple event sources.
- -d : Disable
- -l : File prefix of trace files.

We use the function **PVFS_mgmt_setparam_all** with the corresponding type (*PVFS_SERV_PARAM_EVENT_ENABLE* or *PVFS_SERV_PARAM_EVENT_DISABLE*) to transfer the parameters to the server statemachine **src/server/setparam.sm**. According to the type, new functions which use the HDTrace API in **src/common/misc/pint-event-hd.c** will be called. An excerpt is shown in listing 5.3.

Listing 5.3: Controlling tracing

```
/* src/apps/admin/pvfs2-set-eventmask.c */
int main(args, argv)
{
...
ret = PVFS_mgmt_setparam_all(
          cur_fs, &creds,
          PVFS_SERV_PARAM_EVENT_ENABLE,
          &param_value, NULL, NULL);
...
}

/* src/server/setparam.sm */
static PINT_sm_action setparam_work(
```

```
    struct PINT_smcb *smcb, job_status_s *js_p)
{
...
    case PVFS_SERV_PARAM_EVENT_ENABLE:
      ret = 0;
      PINT_HD_event_initialize(
        s_op->req->u.mgmt_setparam.value.u.string_value, name);
      js_p->error_code = ret;
      return SM_ACTION_COMPLETE;
    case PVFS_SERV_PARAM_EVENT_DISABLE:
      PINT_HD_event_finalize();
      js_p->error_code = 0;
      return SM_ACTION_COMPLETE;
...
}
/* src/common/misc/pint-event-hd.h */
int PINT_HD_event_initialize(const char * traceWhat,
                             const char * projectFile);
int PINT_HD_event_finalize(void);
```

5.1.4 Reduce Modifications to PVFS Source Code

In order to make our code less dependent on PVFS, we use macros to intercept PVFS internal parts. The common macros used in this thesis are *HAVE_HDTRACE*, *HAVE_HDRUTL* as defined in **configure.in** (see listing 5.1). Macros like *__PVFS2_SERVER__* and *__PVFS2_CLIENT__* indicate whether the code is part of the client or server API. External clients (e.g MPI programs) can not include header files of PVFS which have complex dependencies, including PVFS macros, otherwise many internal header files must be installed as well. Instead, we have to use extra header files, so that the external clients can use the new tracing functions.

Moreover, using macros allows us to be able to use the same code structure and reduce code dimensions or duplicates. In PVFS the logic of tracing functions for client and server differs just slightly, therefore we can use the same variables or functions with different macros. An excerpt is shown in listing 5.4.

Listing 5.4: Macros providing traceable internals on client and server

```
/* pint-event-hd.c */
static hdStatsGroup * hd_facilityTrace[ALL_FACILITIES];

/* src/common/misc/pint-event-hd.h */
#ifdef __PVFS2_SERVER__
typedef enum {
    BMI, TROVE, FLOW, REQ, BREQ,
...
    ALL_FACILITIES
} HD_Trace_Facility;
#else /* __PVFS2_CLIENT__ */
typedef enum {
```

```
    BMI, FLOW, CLIENT, // without TROVE, REQ and BREQ
...
    ALL_FACILITIES
} HD_Trace_Facility;
#endif /* __PVFS2_SERVER__ */
```

5.2 Tracing I/O Details and Creation Correlation

In order to trace details of I/O operations the following attributes are needed: *Total Size*, *Size* and *Offset* (see section 4.3.4). For *Total Size* the relevant server statemachines are I/O and small I/O statemachines. In this section we discuss how to trace *Total Size* for normal I/O because we need some tricks to transfer the value of *Total Size* from the I/O statemachine to the corresponding trace function in comparison to small I/O. *Size* and *Offset* are traced in conjunction with the correlation between Server and Trove. Tracing *Size* and *Offset* of a low level I/O operation is not so difficult since these attributes are given at the I/O operations in Trove. In addition, the creation of correlations by using hints is described (see section 4.3.4 for design overview).

5.2.1 Total Size for I/O

Total Size can be found in the function **io_cleanup** of **src/server/io.sm**. This function cleans up the environment. In order to use this attribute for the statemachine structure in **scr/common/misc/state-machine-fns.c** , we have to save it in the data structure *PINT_server_io_op* of **src/server/pvfs2-server.h** before the information about Flow is freed from memory. Therefore, we create a new variable in the structure and set its value to the *Total Size*. An excerpt of the modifications is shown in listing 5.5.

Listing 5.5: Total Size

```
/* src/server/pvfs2-server.h */
struct PINT_server_io_op
{
    flow_descriptor* flow_d;
#ifdef HAVE_HDTRACE
    PVFS_size io_size;
#endif
};

/* src/server/io.sm */
static PINT_sm_action io_cleanup(
    struct PINT_smcb *smcb, job_status_s *js_p)
{
...
    if (s_op->u.io.flow_d)
    {
#ifdef HAVE_HDTRACE
        s_op->u.io.io_size = s_op->u.io.flow_d->file_data.fsize;
```

```
#endif
        PINT_flow_free(s_op->u.io.flow_d);
    }
...
}
```

5.2.2 Correlation between Server and Trove

In this thesis the following two Trove methods are instrumented for showing the correlation between request and low level I/O: *alt-aio* and *directio*. In the method *directio* the hint is already given to the functions which do I/O operations (p_read/p_write). In comparison to *alt-aio* this method can be easily instrumented. Hence, in this section we will discuss how to transfer the hint in *alt-aio*.

Trove method: alt-aio

The method *alt-aio* uses the standardized POSIX AIO interface for submitting I/O operations. The structure is defined in system header files and not part of PVFS. Therefore, it is not easy to transport the additional hint to the alternative implementation. States of processing I/O are shown in figure 5.1. Atomic steps and design considerations are described as below:

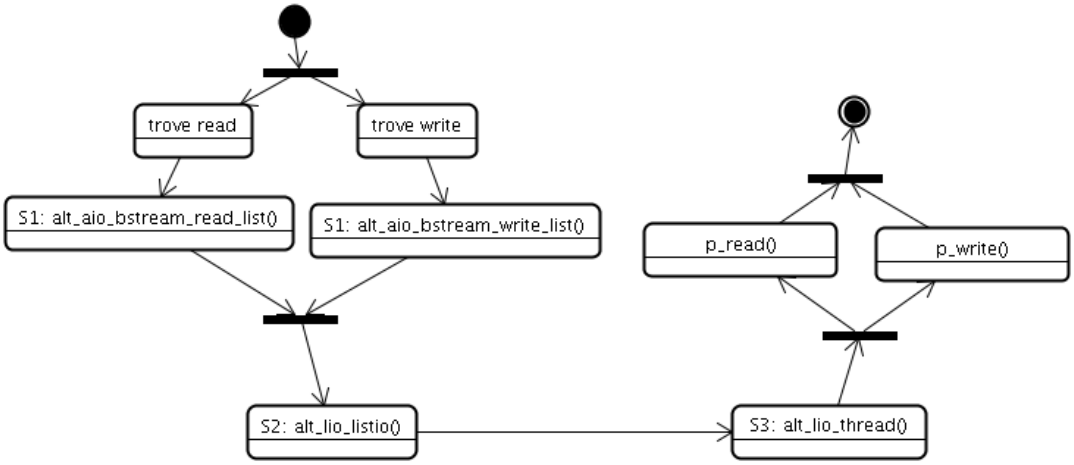

Figure 5.1: Excerpt and modifications of atomic operations for the Trove method alt-aio

S1 To transfer the hint from *alt_aio_bstream_read_list* or *alt_aio_bstream_write_list* to *p_read/p_write*, in brief we need to transfer it via *alt_lio_listio* and *alt_lio_thread*. There is an array, which consists of metadata of I/O operations like *Size* and *Offset*, and is used in many steps between *alt_aio_bstream_read_list* or *alt_aio_bstream_write_list* to *alt_lio_listio*. The array uses a macro for its size (AIOCB_ARRAY_SZ), hence we can increase the array by one (AIOCB_ARRAY_SZ + 1) and put a pointer to the hint into it. Note we have to make the changes wherever the array is declared.

```
/* src/io/trove/trove-dbpf/dbpf-alt-aio.c */
static int alt_lio_listio(int mode, struct aiocb * const list[],
                          int nent, struct sigevent *sig);

/* src/io/trove/trove-dbpf/dbpf-bstream.c */
// increase the array size
#ifdef HAVE_HDTRACE
struct aiocb * aiocb_ptr_array[AIOCB_ARRAY_SZ+1];
#else
struct aiocb * aiocb_ptr_array[AIOCB_ARRAY_SZ];
#endif
// put the hint into the array
#ifdef HAVE_HDTRACE
aiocb_ptr_array[aiocb_inuse_count] = (struct aiocb *)
                                     cur_op->op.hints;
#endif
```

S2 To transfer the hint from *alt_lio_listio* to *alt_lio_thread*, we need to create a new variable for the hint in the data structure *alt_aio_item* and save the hint from the array discussed above.

```
/* src/io/trove/trove-dbpf/dbpf-alt-aio.c */
struct alt_aio_item
{
...
    // create a new entry for hint
#ifdef HAVE_HDTRACE
    PVFS_hint hints;
#endif
};

// set the hint value in alt_aio_item
static int alt_lio_listio(int mode, struct aiocb * const list[],
                          int nent, struct sigevent *sig);
{
...
// take the hidden hint
#ifdef HAVE_HDTRACE
PVFS_hint hints = (PVFS_hint) list[nent];
tmp_item->hints = hints
#endif;
...
ret = pthread_create(&master_tid, &attr, alt_lio_thread,
                     tmp_item);
...
}
```

S3 The function *alt_lio_listio* calls the function *alt_lio_thread* with the parameter *tmp_item* containing the hint. Now we take the hint out and use it to trace the function call to *p-read/p-write*, so the work is done.

```
/* src/io/trove/trove-dbpf/dbpf-alt-aio.c */
static void* alt_lio_thread(void* foo)
{
  struct alt_aio_item* tmp_item = (struct alt_aio_item*) foo;
  if(tmp_item->cb_p->aio_lio_opcode == LIO_READ)
  {
    HD_SERVER_TROVE_RELATION(tmp_item->hints,"alt-io-read",
        ret = pread(tmp_item->cb_p->aio_fildes,
                    (void*)tmp_item->cb_p->aio_buf,
                    tmp_item->cb_p->aio_nbytes,
                    tmp_item->cb_p->aio_offset);,
                    "%d",tmp_item->cb_p->aio_nbytes,
                    "%lld",lld(tmp_item->cb_p->aio_offset)
    )
  }
...
}

/* src/common/misc/pint-event-hd.h */
#define HD_SERVER_TROVE_RELATION(hints, name, CMD, p_size, size,
                                 p_offset, offset) \
  const char * io_keys[] = {"size","offset"}; \
  char attr1[15],attr2[15]; \
  char * io_values[] = {attr1, attr2}; \
  const char * c_io_values[2]; \
  int run = 1; \
  HD_SERVER_TROVE_RELATION(SERVER, \
    hdR_token relateToken = NULL; \
      hdR_token parentToken = *(hdR_token*)  \
      PINT_hint_get_value_by_name(hints, \
          PVFS_HINT_RELATION_TOKEN_NAME, NULL); \
        \
        if (parentToken && topoTokenArray[TROVE]) \
        { \
          gen_mutex_lock(&trove_relation); \
          relateToken = hdR_relateProcessLocalToken \
          (topoTokenArray[TROVE], parentToken); \
          gen_mutex_unlock(&trove_relation); \
        } \
        \
        if (relateToken) \
        { \
          gen_mutex_lock(&trove_relation); \
          hdR_startS(relateToken,name); \
          gen_mutex_unlock(&trove_relation); \
          run = 0; \
          \
          CMD \
          \
```

```
            snprintf(io_values[0], 15, p_size, size); \
            snprintf(io_values[1], 15, p_offset, offset); \
            c_io_values[0] = io_values[0]; \
            c_io_values[1] = io_values[1]; \
            \
            gen_mutex_lock(&trove_relation); \
            hdR_end(relateToken,2,io_keys,c_io_values); \
            hdR_destroyRelation(&relateToken); \
            gen_mutex_unlock(&trove_relation); \
          } \
) \
if(run){ \
        CMD \
} \

#define HD_SERVER_RELATION(facility, stmt) \
  do{ if(topoTokenArray[facility]){ stmt } } while(0);
```

Summary: According to design considerations in chapter 4, some complex modifications for the implementation have been explained. Moreover, the whole build process of PVFS was explained for the code integration. The macros are defined in the file **configure.in** and used to instrument PVFS internals. In other words, the changes are kept optional by triggering the tracing only via parameters during source configuration.

Next, experiments made with these modifications are discussed.

6 Evaluation

In this chapter some basic features of Sunshot are introduced. Information of the cluster configuration is also provided. There are two experiments made in this thesis. The first experiment is made on a balanced system. In conjunction with this experiment new metrics and tracing information in Sunshot are discussed in detail, so that the users can understand the visualization. In contrast, the second experiment is made on an unbalanced system. In this experiment we will analyze factors which are related to bottlenecks.

6.1 Basic Features of Sunshot

This section highlights some basic features in Sunshot for the understanding of results made in this thesis. An example is shown in figure 6.1 and described as follows:

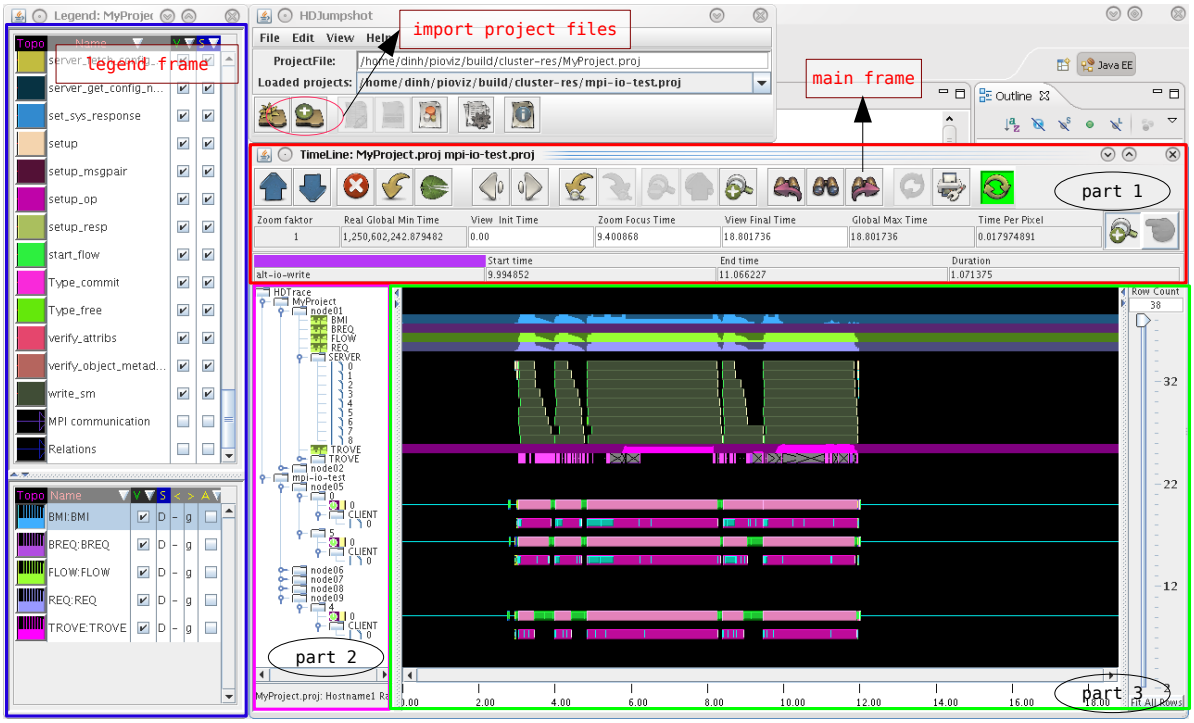

Figure 6.1: Basic Sunshot feature

Import multiple project files: Due to the new trace format of HDTrace Sunshot permits a visualization of client and server processes by importing different project files.

Main frame:

- Part 1: The info panel contains meta information of the event located under the mouse pointer, including start time, end time, duration, zoom factor, value, etc.
- Part 2: contains the potentially modified topology tree viewed in Sunshot according to project files.
- Part 3: contains the visualization of the topology events/statistics with many time lines according to the components.

Legend frame: In this frame users can choose the events and statistics to be viewed in the main frame. Additional parameters for the visualization of statistics can be specified. Right now relations can be added in this frame as well.

6.2 Cluster Configuration

The experiments were performed on the working group's cluster with nine nodes. Each node is equipped with two Intel Xeon 2 GHz processors with 512 KB Cache, 1 GB RAM, 80 GB IBM HDD, an Intel 82545EM Gbit Ethernet and Network Interface. The Ubuntu server edition 8.04 is installed on the cluster. PVFS (version 2.8.2) was installed on four nodes (as servers). One server is used as a metadata and data server; the others as data servers only. The remaining five nodes are used for MPI client programs with MPICH2 (version 1.08p1) installed.

6.3 Experiments

In this section we show two experiments and discuss the results with the benchmark program **mpi-io-test** shipped with ROMIO. The test case was to write 10 times 100 MB with 9 client processes:

```
ssh node05 $HOME/pioviz/bin/mpiexec -np 9 -env PVFS2TAB_FILE $PVFS2TAB_FILE
-env PVFS2_HD_TRACE_CLIENT $PVFS2_HD_TRACE_CLIENT $HOME/MPI-IO/mpi-io-test
-p $HOME/pioviz/pvfs2tab -i 10 -v -w -b $((10*1024*1024))
-f pvfs2://pvfs2/example
```

The first experiment is conducted in a balanced environment consisting of identical hardware and software condition. In this experiment we will discuss the new tracing metrics and information in detail. In the second experiment background processes which stress the system are started on a particular server. We will discuss and analyze whether we can localize the bottlenecks.

6.3.1 Demonstrating New Traced Information

The screenshot of Sunshot is shown in figure 6.2. For a good view, we keep one server (node01) with one client (node05) like in figure 6.3. Based on this figure the results will be discussed in detail.

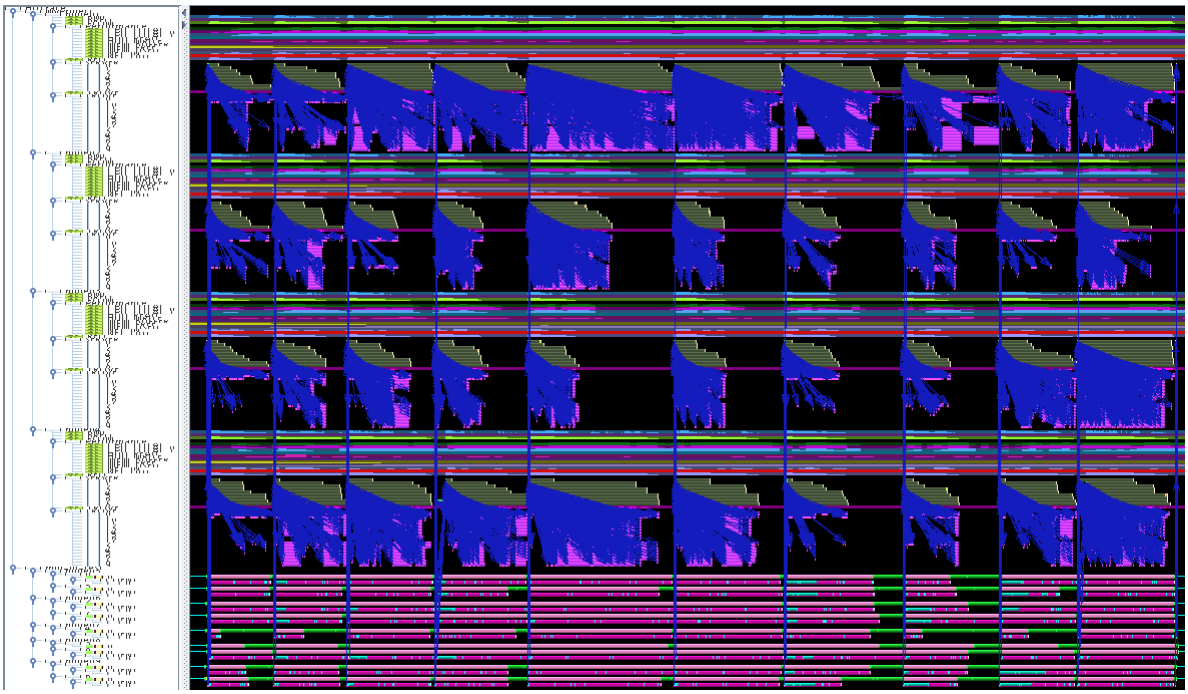

Figure 6.2: Five clients and four servers

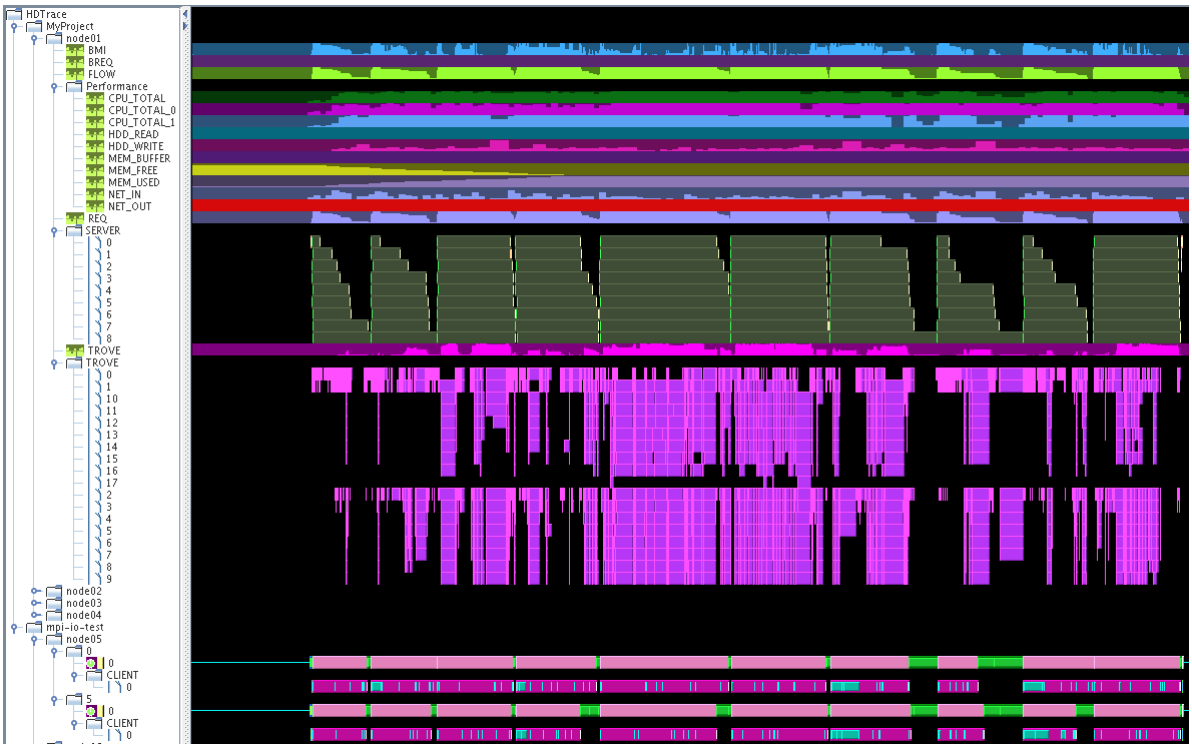

Figure 6.3: One client and one server

Hardware Statistics and Number of concurrent Operations

The two metrics, hardware statistics and the number of concurrent operations, are highlighted in figure 6.4. We will discuss and analyze a few regions of the trace. Therefore, the marked areas are made and the discussion is based on these areas. Number in statistics timelines (in black) indicate the maximum value observed in the region.

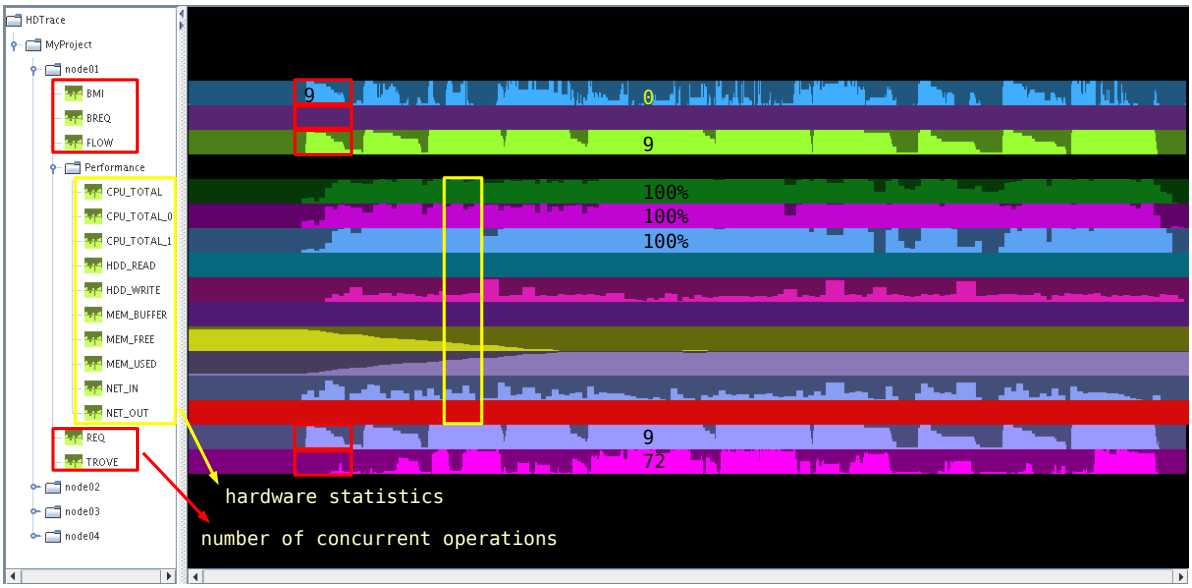

Figure 6.4: Hardware statistics and number of concurrent operations

Hardware Statistics: In case of the write operation with 1 GB, we can verify the correctness of the hardware statistics, for instance HDD_READ is fast zero and HDD_WRITE is not. In the marked area the trend of HDD_WRITE (values) increases, that's why CPU_TOTAL is high, MEM_USED increases and MEM_FREE decreases. Because no data is transferred to the clients, NET_OUT is very low. It is difficult to compare NET_IN with HDD_WRITE because data is cached before being written to the hard disks, thus we see that these two areas look different. In comparison to the other metric, the values of this metric stay constant for some time because these values are updated in a fixed time interval.

Number of concurrent Operations: The values of the marked areas for BMI, Flow and REQ do not look exactly the same but similar because normally a operation in REQ will trigger operations in Flow, a operation in Flow triggers operations in BMI and Trove, and on the other hand a operation in BMI triggers operations in Flow (see figure 2.1.1). It is reasonable that the values of Breq are zero because no metadata operations occur during write requests. We may think, an error occurred in Trove because at the beginning only a few operations are visible. That is not the case, due to write behind data is cached in the fast memory and therefore operations finish quickly. Write-back from memory to disks is deferred. Later, the cache get filled, then the OS must block further write operations issued by Trove. Therefore, the observed values in Trove increase with decreased cache. For the whole time line of each layer we can see that the value of this metric begins with zero, then changes (increases or decreases) and will be again zero at the end.

Observation: For the special case where all the CPUs are maximum 100% loaded in node01 (marked with numeric values), we see that the values of BMI are zero, the maximum values of Flow and REQ are nine. In PVFS, the data of each process will be split and transferred in eight small parts successively,

each with 256 KB. After a part is done (read/written) in Trove, the next part can be transferred by BMI. In this case all eight buffers are received, i.e. BMI does not need to receive further data. For each part there is a corresponding I/O thread started in Trove. In other words, each Flow request issues eight Trove operations. So in this case the number of concurrent Trove operations is 72 (8 x 9 = 72).

Statemachine Tracing

Figure 6.5 shows how nested states of a statemachine can be viewed in Sunshot. Thus, we can understand internal processing of particular operations in detail. In this figure, the arrows indicate the names of states viewed under the mouse pointer.

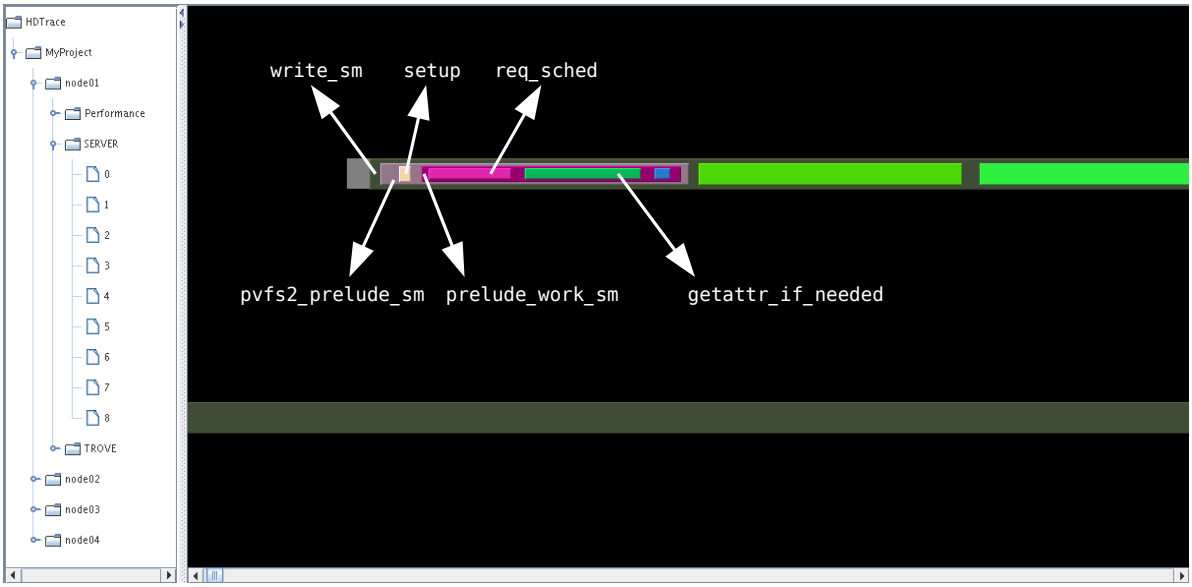

Figure 6.5: Nested states and statemachines

Figure 6.6 shows an pattern of state overlapping in Sunshot when multiple operations run concurrently. Figure 6.7 is an screenshot of expanded time lines from figure 6.6.

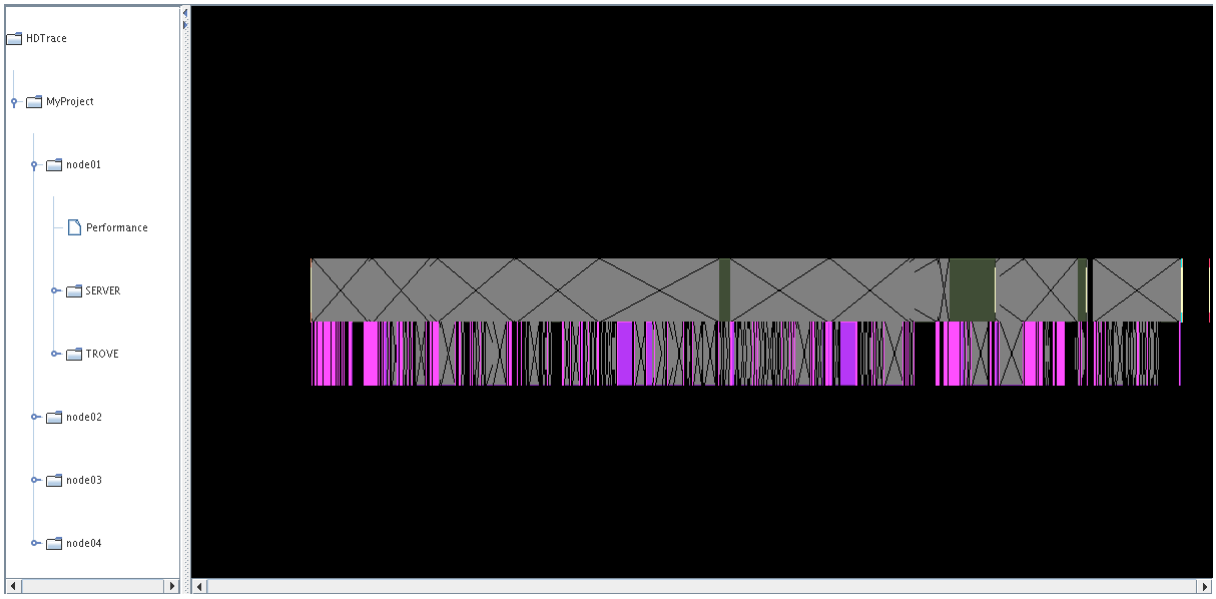

Figure 6.6: State overlapping with one time line

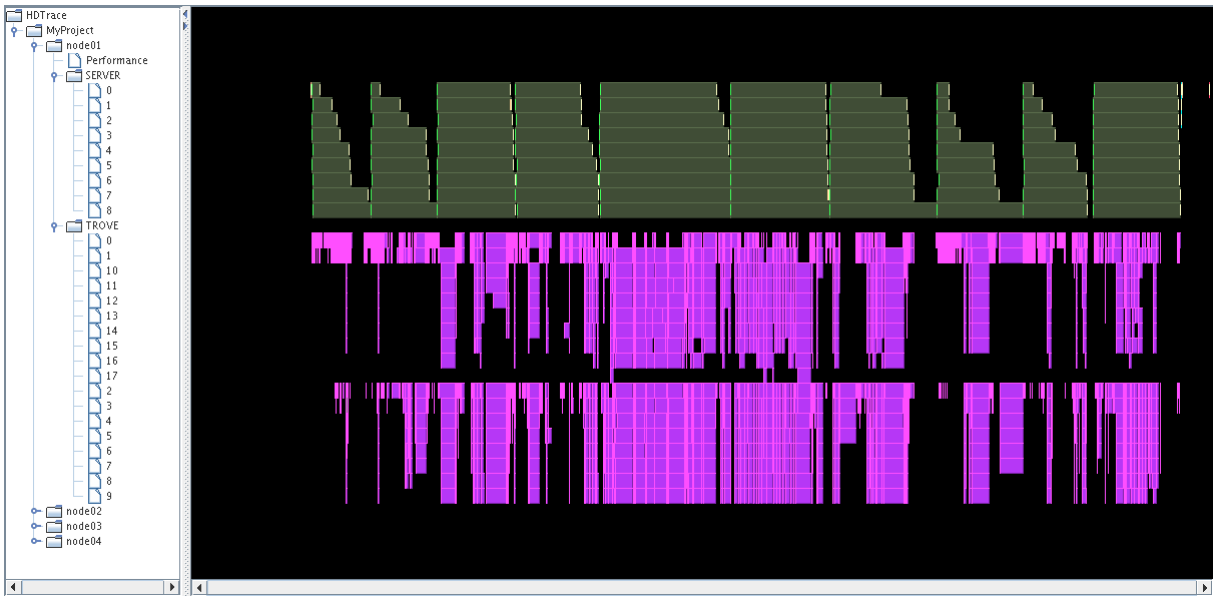

Figure 6.7: Expanded time lines with multiple time lines

Correlation Creation

In figure 6.8 arrows indicate which client operations triggered which server operations and similarly which server operations triggered which Trove operations. Thereby, their correlations are created. However, information in this figure is overwhelming for the users because the information is not well presented in Sunshot. For instance, if there are many operations traced, it is really hard or even impossible to keep track the call history, i.e. which operations called which operations, although the arrows for the correlations are created. Therefore, Sunshot should and will be extended to support the users in filtering the context specific information.

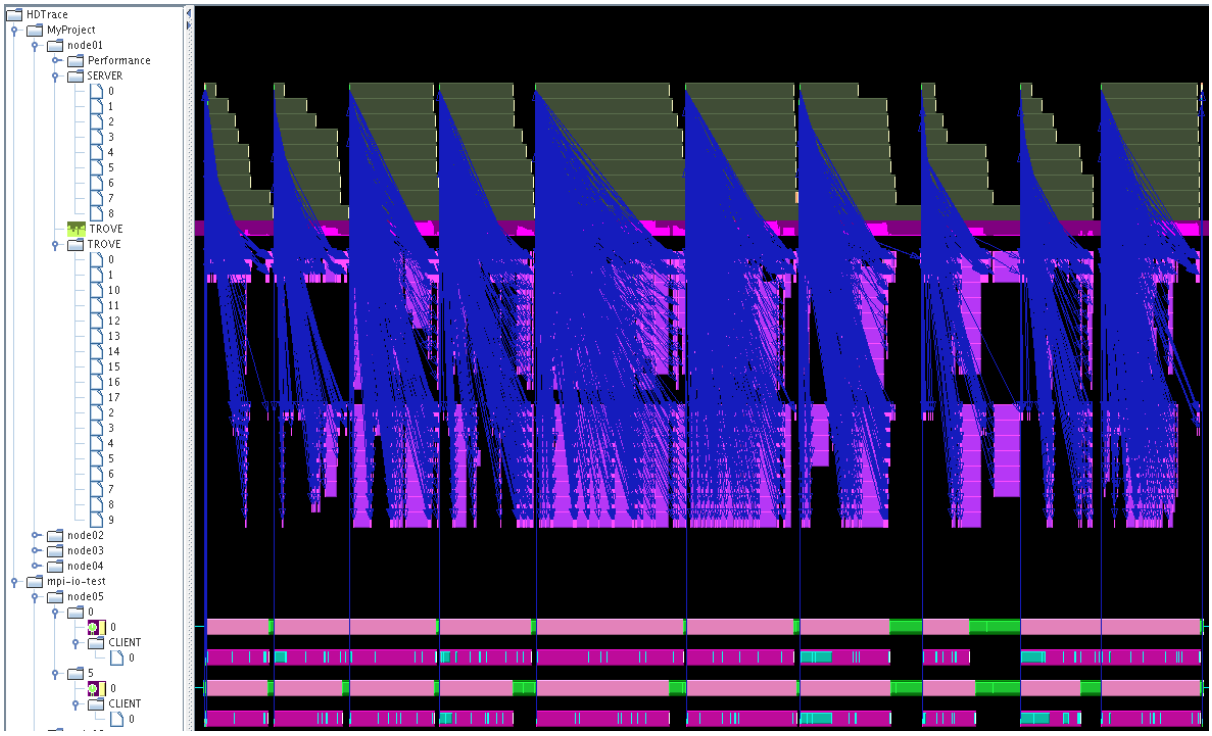

Figure 6.8: Correlations

Tracing I/O Statistics:

In figure 6.9 we can see the value of *Total Size* of each operation of a server process and in addition its thread operations in Trove via *Size* and *Offset*. Moreover, the type of I/O method and other meta data are collected as well.

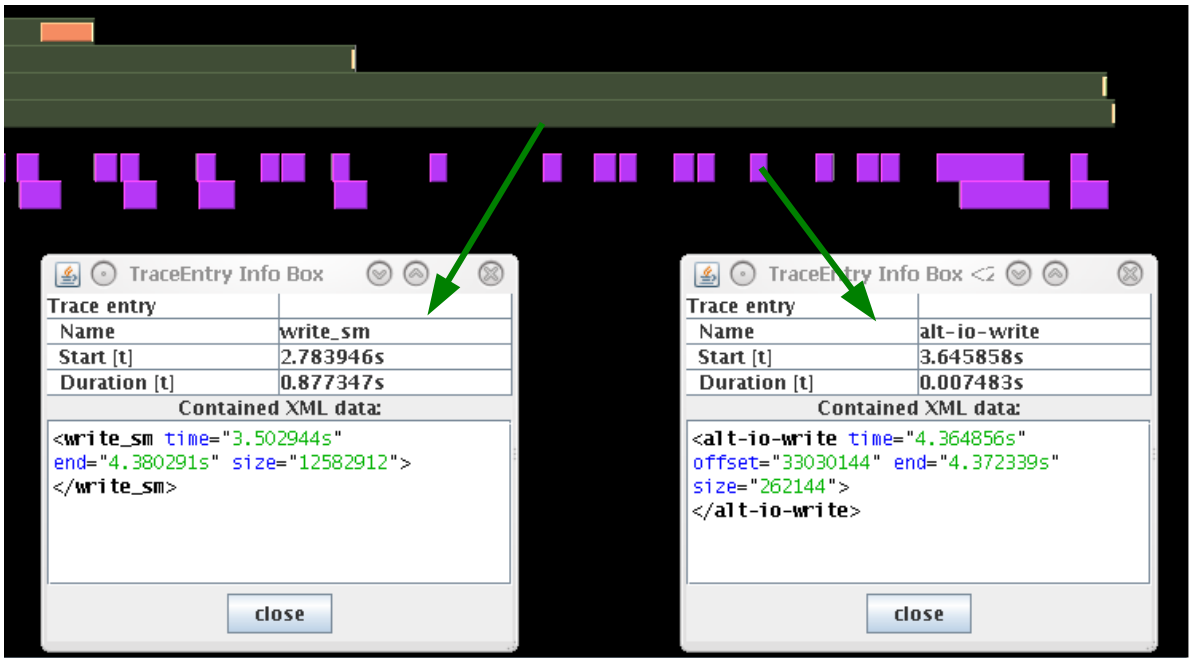

Figure 6.9: I/O statistics

6.3.2 Analyzing Unbalanced Server Condition

In this experiment one server (node02) gets stressed (both CPU 100% load, reducing free memory to 200MB) by starting two background process. The screenshot is shown in figure 6.10. At first sight we see that the results on node02 differ from those on other servers. Results on other nodes are similar, for simplicity we will discuss the differences between node01 and node02.

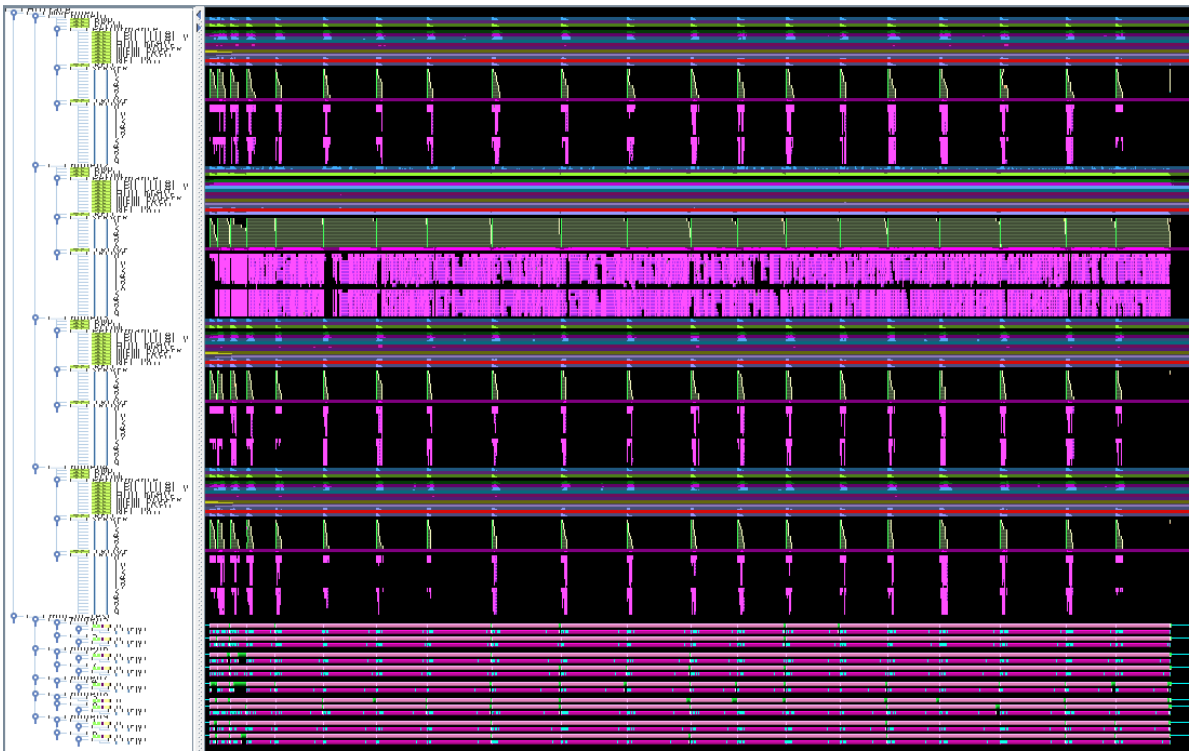

Figure 6.10: Five clients and four servers

Hardware Statistics and Number of concurrent Operations

An excerpt of figure 6.10 is shown in figure 6.11.

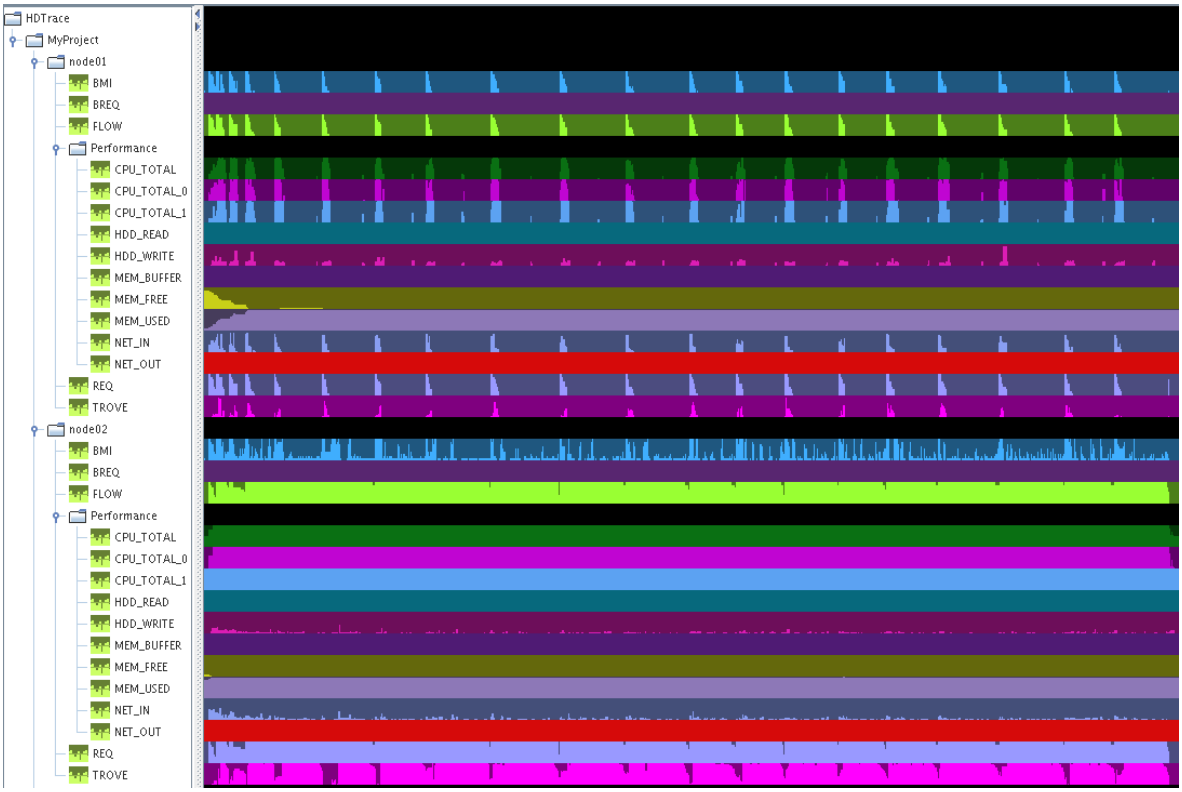

Figure 6.11: Hardware statistics and concurrent operations

It is easy to recognize the significant difference between components of these two nodes. In comparison to node01 the values of HDD_WRITE in node02 are high all the time, and at the beginning MEM_FREE is lower. This means the values are as expected by this experiment. Now we will analyze how these differences influence performance of other components in PVFS.

Due to the low resource of node02 there are many pending operations. The comparison with integrated average values is shown in table 6.1.

Components	**Node01**	**Node02**
BMI	0.462	0.771
Flow	0.677	7.652
REQ	0.680	7.656
Trove	1.456	54.433
NET_IN	3185 (KB)	3589 (KB)
HDD_WRITE	6695 (Blocks)	6411 (Blocks)

Table 6.1: Comparison of BMI, Flow, REQ, Trove, NET_IN and HDD_WRITE

According to the comparison the values of other information, especially Trove of node02 are significantly higher than of node01. In other words, the environment in this experiment is unbalanced.

Figure 6.12 indicates that there are many concurrent operations in Trove and they run longer due to the low resource availability in node02.

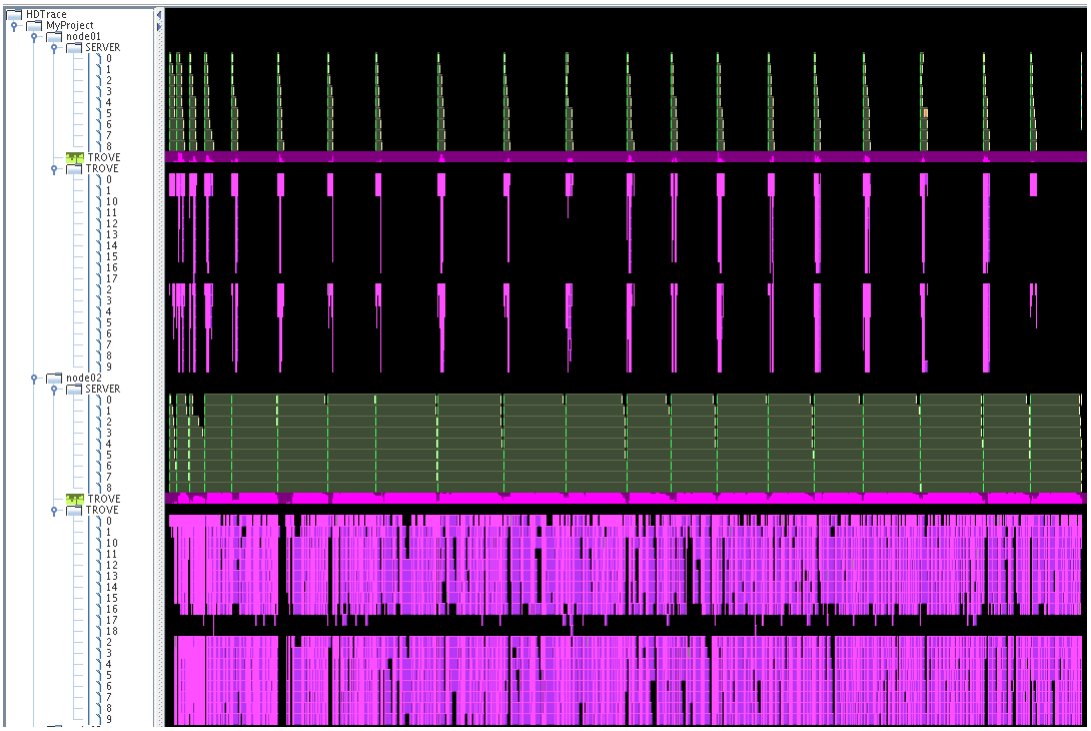

Figure 6.12: States in Trove

The reasons are: in this experiment the method *alt-aio* is used. This method spawns I/O threads. Thus, it is CPU intensive. Probably, it slows the real I/O. In node02 the I/O cache is much smaller than in node01. Thereby, on node02 the merging of the write operation is ineffective, and the threads blocked by the operating system take time much longer, as we can see in the figure 6.12. For example, on node02 there are many threads with the time duration over 9 seconds. It is too long for a write operation and indicates a bottleneck due to write block.

Summary: Basically, there were two experiments presented in this chapter. The first experiment was made on a balanced environment. In this experiment, results including new metrics and tracing information were discussed in detail via Sunshot. In contrast, the second test was made on an unbalanced environment. Overall we see that the new metrics and tracing information are useful to track bottlenecks down. However, the information is not presented optimal by Sunshot so far. Thereby, in the future, Sunshot should provide more features to help the users to deal with the information.

7 Summary, Conclusion and Future Work

7.1 Summary and Conclusion

In this thesis relevant performance metrics and information is defined to localize bottlenecks. These metrics are implemented into the parallel file system PVFS to trace client and server performance behavior. In detail modifications are as follows:

- *Hardware Statistics*: In comparison to the statistics provided by PIOViz 1.0, some new statistics are added. They are more powerful and contain many needed information for the analysis. In PIOViz 1.0 we got only one statistic for CPU. Now it is fine grained, which means each CPU has a statistic, and it is much more useful for the analysis.
- *Number of concurrent operations*: To have an overview of the system it is important to know how many operations in PVFS internal layers and statemachines are running. This metric also can be used to compare the load between different nodes.
- *Statemachine tracing*: Statemachines are the fundamental concept in PVFS. Each statemachine consists of many states, each has a state action function, and can be nested with other statemachines. Tracing this information is useful to understand and manage operations within the process.
- *Correlations creation*: Beside of factors related to bottlenecks, it is also important to understand the correlations between layers in PVFS in order to determine the reason for the system behavior, e.g. which client caused the inefficient I/O operations in Trove.

Modifications made to PVFS use the new tracing API HDTrace of PIOViz to store system events in trace files. Then they can be viewed and analyzed via the visualization tool Sunshot to assist performance analysis.

Furthermore, the changes made in PVFS are activated by the build process via parameters during configuration. This allows to check the overhead imposed by the modifications.

With the new PIOViz environment including HDTrace and Sunshot as well as metrics or tracing information made in this thesis it is easier for users to analyze the system and detect bottlenecks. We have seen one experiment where one server is stressed a lot by a particular program. Sunshot showed where bottlenecks had occurred and which layers or components were related.

7.2 Future Work

Currently, the correlations between client/server and server/Trove are created. However, it also might be useful to trace the correlations between other layers in PVFS (see section 4.3.4).

However, the interpretation of information is a bit difficult for the users because it is not arranged very well in Sunshot. Therefore, the information should be arranged in a simple way, for example, a context based guide to help the users. These extensions are already planned for Sunshot and possible due to the new trace capabilities.

Right now, the server project file (see section 4.4) must be created manually. Besides the client project file must be manually adjusted for PVFS internal tracing. In the future, the project description merger should be extended to overcome this problem.

In this thesis there are two simple experiments made. We need to perform more experiments to assess metrics defined and continue to find new interesting metrics and trace information related to bottlenecks.

Bibliography

[Kuh07] Michael Kuhn. Directory-Based Metadata Optimizations for Small Files in PVFS. Bachelor's thesis, Ruprecht-Karls-Universität Heidelberg, Institute of Computer Science, Research Group Parallel and Distributed Systems, September 2007.

[Kun07] Julian Martin Kunkel. Towards Automatic Load Balancing of a Parallel File System with Subfile Based Migration. Master's thesis, Ruprecht-Karls-Universität Heidelberg, Institute of Computer Science, Research Group Parallel and Distributed Systems, August 2007.

[LKK+06] Thomas Ludwig, Stephan Krempel, Julian M. Kunkel, Frank Panse, and Dulip Withanage. Tracing the MPI-IO Calls' Disk Accesses. In Mohr et al. [MTWD06], pages 322–330.

[LKK+07] Thomas Ludwig, Stephan Krempel, Michael Kuhn, Julian Kunkel, and Christian Lohse. Analysis of the MPI-IO Optimization Levels with the PIOViz Jumpshot Enhancement. page 2, 2007.

[MTWD06] Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Proceedings*, volume 4192 of *Lecture Notes in Computer Science*. Springer, 2006.

[Paga] MPICH2 Home Page. MPICH2. Homepage `http://www.mcs.anl.gov/research/projects/mpich2`.

[Pagb] Paradyn Home Page. Paradyn. Homepage `http://www.cs.wisc.edu/paradyn`.

[Pagc] Vampir Home Page. Vampir. Homepage `http://www.vampir.eu/index.html`.

[Proa] The PVFS2 Project. Parallel Virtual File System. Homepage `http://pvfs.org`.

[Prob] The PVFS2 Project. Parallel Virtual File System Documentation. Homepage `http://pvfs.org/cvs/pvfs-2-8-branch-docs/doc//pvfs2-guide.pdf`.

[Tea03] PVFS Development Team. Parallel Virtual File System Version 2. PVFS2 Internal Documentation included in the source code package, 2003.

[TGL96] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *FRONTIERS '96: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, page 180, Washington, DC, USA, 1996. IEEE Computer Society.

[TGL99] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1999.

A Appendix

A.1 PVFS Installation

This section describes how to install PVFS on four nodes like the experiment made in section 6.2.

```
# Configure and install the packages
$ tar xvf pvfs-2.8.1.tar.gz
$ mkdir $PWD/installed
$ mkdir $PWD/storage
$ cd pvfs-2.8.1
$ configure --prefix=$PWD/installed
$ export INSTALL=$PWD/installed
$ export STORAGE=/tmp/pvfs2-storage
$ make
$ make install

# Create the configuration file
$ pvfs2-genconfig fs.conf --protocol=tcp --tcpport=5001 --ioservers=
node01,node02,node03,node04 --metaservers=node01 --storage=$STORAGE
--logfile=$STORAGE/log

# Create storage
$ $INSTALL/sbin/pvfs2-server fs.conf -f

# Create the file pvfs2tab with the content and set the environment
for PVFS2TAB_FILE
echo "tcp://node01:5001/pvfs2-fs /pvfs2 pvfs2 defaults,noauto 0 0" \
> $PWD/pvfs2tab
export PVFS2TAB_FILE=$PWD/pvfs2tab
```

A.2 PVFS File System Configuration Files

A.2.1 One Metadata and Four data servers

```
<Defaults>
        UnexpectedRequests 50
        EventLogging none
        EnableTracing no
        LogStamp datetime
```

```
        BMIModules bmi_tcp
        FlowModules flowproto_multiqueue
        PerfUpdateInterval 1000
        ServerJobBMITimeoutSecs 30
        ServerJobFlowTimeoutSecs 30
        ClientJobBMITimeoutSecs 300
        ClientJobFlowTimeoutSecs 300
        ClientRetryLimit 5
        ClientRetryDelayMilliSecs 2000
        PrecreateBatchSize 512
        PrecreateLowThreshold 256

        StorageSpace /tmp/pvfs2-storage
        LogFile /tmp/pvfs2-storage/log
</Defaults>

<Aliases>
        Alias node01 tcp://node01:5001
        Alias node02 tcp://node02:5001
        Alias node03 tcp://node03:5001
        Alias node04 tcp://node04:5001
</Aliases>

<Filesystem>
        Name pvfs2-fs
        ID 911008748
        RootHandle 1048576
        FileStuffing no
        <MetaHandleRanges>
                Range node01 3-1844674407370955162
        </MetaHandleRanges>
        <DataHandleRanges>
                Range node01 1844674407370955163-3689348814741910322
                Range node02 3689348814741910323-5534023222112865482
                Range node03 5534023222112865483-7378697629483820642
                Range node04 7378697629483820643-9223372036854775802
        </DataHandleRanges>
        <StorageHints>
                TroveSyncMeta yes
                TroveSyncData no
                TroveMethod alt-aio
        </StorageHints>
</Filesystem>
```

A.2.2 Server Configuration

The configuration for each server (node01-node04) looks similar - only the hostname is changed, e.g. node02 instead of node01.

```
StorageSpace /tmp/pvfs2-storage/
HostID "tcp://node01:5001"
LogFile /tmp/pvfs2-storage/log
```

A.3 Start PVFS

```
$ for I in `seq 1 4`; do \
ssh node0$I "$INSTALL/sbin/pvfs2-server fs.conf" \
done
```

A.4 Stop PVFS

```
$ for I in `seq 1 4`; do \
ssh node0$I "killall pvfs2-server" \
done
```

A.4.1 An Example of a Server Project File

```
<?xml version="1.0" encoding="UTF-8"?>
<Application name="result" processCount="3">
<Description>normal description </Description>

<FileList>
<File name="pvfs2://pvfs2//visualization.dat">
  <InitialSize>0</InitialSize>
  <Distribution class="de.hd.pvs.piosim.model.inputOutput.
   distribution.SimpleStripe">
  <ChunkSize>64K</ChunkSize>
  </Distribution>
</File>
<File name="pvfs2://pvfs2//checkpoint.1">
  <InitialSize>0</InitialSize>
  <Distribution class="de.hd.pvs.piosim.model.inputOutput.
   distribution.SimpleStripe">
  <ChunkSize>64K</ChunkSize>
  </Distribution>
</File>
<File name="pvfs2://pvfs2//checkpoint.2">
  <InitialSize>0</InitialSize>
  <Distribution class="de.hd.pvs.piosim.model.inputOutput.
   distribution.SimpleStripe">
  <ChunkSize>64K</ChunkSize>
  </Distribution>
</File>
</FileList>

<Topology>
        <Level type="Hostname1">
        <Level type="Rank1">
        <Level type="Thread1">
        </Level>
        </Level>
        </Level>
```

```
        <Node name="node01">
                <Node name="SERVER" />
                <Node name="TROVE" />
                <Node name="CLIENT" />
        </Node>
        <Node name="node02">
                <Node name="SERVER" />
                <Node name="TROVE" />
        </Node>
        <Node name="node03">
                <Node name="SERVER" />
                <Node name="TROVE" />
        </Node>
        <Node name="node04">
                <Node name="SERVER" />
                <Node name="TROVE" />
        </Node>

</Topology>

<CommunicatorList>
<Communicator name="WORLD">
<Rank global="0" local="0" cid="1"/>
</Communicator>
</CommunicatorList>

<ExternalStatistics>
<BMI/>
<FLOW/>
<TROVE/>
<REQ/>
<BREQ/>
<Performance/>
</ExternalStatistics>

</Application>
```

A.5 Tools used for the Thesis

```
Operating system: Ubuntu 9.04
Pdf Editor: Kile, Latex
Image and Diagram: Open Office, Jude, ksnapshot
IDE: Eclipse 3.4
```

List of Figures