Ruprecht-Karls Universität Heidelberg
Institute of Computer Science
Research Group Parallel and Distributed Systems

Internship Report

Programmable I/O-Pattern Benchmark for Cluster File Systems

Name: Dennis Runz, Christian Seyda
Tutors: Olga Mordvinova, Julian M. Kunkel

Summer Term 09
April 1, 2009 - September 1, 2009

Contents

1 Introduction

The main goal of this practical was to develop a benchmark for cluster file systems, where custom I/O-patterns can be passed to. To accomplish this, we designed an interpreter based benchmark. We use Flex [1] and Bison [2] to realize the scanner/parser layer for this approach. We also installed two modern cluster file systems, GlusterFS [3] and Ceph [4], to keep up with current technologies.

This document should explain the structure of the benchmark enough to understand the internals and to be able to work with the source code.

1.1 Motivation

Currently, there is still a lack of up to date benchmarks offering real I/O-patterns from real applications. Either the tools use old or synthetic patterns for benchmarking, or they didn't serve all necessary features. To close this gap we developed an open source benchmark tool, which satisfies those requirements. The decision for open source enables the possibility to involve the community, which can also create and contribute I/O-patterns of state-of-the-art applications.

The document is structured as follows: section 2 gives and overview of the benchmark, with feature specification and implementation details. Section 3 describes the installation of the file systems Ceph and GlusterFS on the test cluster. Finally, we arrive at a conclusion about our work.

2 I/O-Pattern Benchmark

2.1 Design Goals

We decided to develop the benchmark in the C programming language as well as using MPI for the parallel functionality, because the target was to make parallel I/O on cluster like environments where C and MPI is widely spread.

Since we wanted to create a tool which enables a high range of flexibility, we decided to design an own programming language. Thus, we needed a parsing and compiling layer. For usability reasons we decided to use an interpreter model. Since efficient lexical analysis and grammar processing are challenging, we decided to use Flex and Bison. They are the open source implementations of Lex and Yacc [5]. Those tools generate fast and reliable C code which does the lexical scanning and grammatical parsing [1], each takes a config file which describes keywords and grammar rules. Because of that we are able to easily extend and modify the programming language grammar as well as increase the feature richness of the benchmark. This concept made it possible to get a powerful parsing layer with less effort.

Additionally, we needed efficient data structures to accomplish the interpreter model. The Gnome Library (`glib2`) [6] offers a powerful API with lots of different data structures as well as - which is indeed the core application - a handy POSIX I/O API which serves us for all sequential I/O needs. Next to this we also wanted to have real parallel I/O. To accomplish this, we use MPI I/O.

2.2 Specification

We differ three kinds of language constructs: basic control flow constructs, sequential I/O constructs and parallel I/O constructs.

[1] $LALR(1)$ - One token lookahead, left to right, right reduce to start symbol (shift/reduce, bottom up parser)

2.2.1 Basic Language Constructs

Two basic keywords which play a important role are `repeat`, which is to execute code blocks in a loop, and `time`, which allows time measurement of every command supported by the programming language. These commands have a special syntactic role, because they can be used as a prefix of every other command as well as using a code block. Listing 2.1 gives an example.

Listing 2.1: Timing and Loops

```
time read(...);
time {
  write(...);
  read(...);
}

repeat 10 read(...);
repeat 10 {
  write(...);
  read(...);
}
```

Furthermore it is possible to use variables, both user defined and internal ones like process rank or random numbers. In the following we list all basic keywords of the programming language:

- `time` - Measurement of execution time
- `repeat` - Repeat a block of code several times
- `print` - Print text to stdout
- `barrier` - Force specified processes to synchronize
- `group` - Group block to limit code execution to certain processes
- `define` - Static defines of data structures
- `groups` - Data structure for groups
- `pattern` - Data structure for patterns

2.2.2 I/O Language Constructs

For simple sequential I/O benchmarking, we cover all necessary POSIX I/O commands:

- `read` - Read file content into memory and free
- `write` - Write given amount of bytes to a file
- `append` - Append given amount of bytes to a file
- `create` - Create a file
- `lookup` - Test if a file exists
- `delete` - Delete a file
- `mkdir` - Create a directory
- `rmdir` - Remove a directory
- `stat` - Read attributes from file or directory
- `rename` - Rename file or directory

We also support explicit file handles where an open and close needs to be issued manually. Thus we defined the following commands, which all share an `f` prefix for file handles:

- `fopen` - Open file and return a handle
- `fclose` - Close file from a handle
- `fread` - Read file content into memory and free (from a handle)
- `fwrite` - Write given amount of bytes to a file (from a handle)

Additionally, for true parallel I/O, we are using MPI I/O and support simple equal size array splitting by using the MPI array type and individual file pointers. Figure 2.1 gives an example with four processes, each gets an equal sized part of the file.

Figure 2.1: File Splitting with 4 Processes

The read and write commands expect a pattern as parameter. It describes how the parallel I/O will be achieved. All parallel I/O commands share a `p` prefix:

- `pread` - Read file content into memory in parallel and free
- `pwrite` - Write given amount of bytes to a file in parallel
- `pfopen` - Collectively open file and return a handle
- `pfclose` - Close file from a handle
- `pfread` - Read file content into memory in parallel and free (from a handle)
- `pfwrite` - Write given amount of bytes to a file in parallel (from a handle)

2.2.3 Parallel Language Constructs

Another fundamental construct for creating more complex test programs is the grouping. It is possible to assign different processes to different user-defined groups, where the process to group mapping can be controlled by the test writer. Groups and mapping rules are defined in the test program and the size of the groups can be set on command line, when starting the program. To influence the mapping it is possible to set tags for each group. This for example allows to have disjoint as well as unique process groups.

Listing 2.2: Grouping

```
define groups {"group1", "group2"};
group "group1" {
  ...
}
```

Parallel I/O will be accomplished by defining a pattern first, then passing it to parallel I/O commands which expect it. A pattern is defined by an identifier name, the number of iterations, the number of elements and the level of parallelism which can be:

- **Level 0:** Non-Collective Contiguous
- **Level 1:** Collective Contiguous
- **Level 2:** Non-Collective Non-Contiguous
- **Level 3:** Collective Non-Contiguous

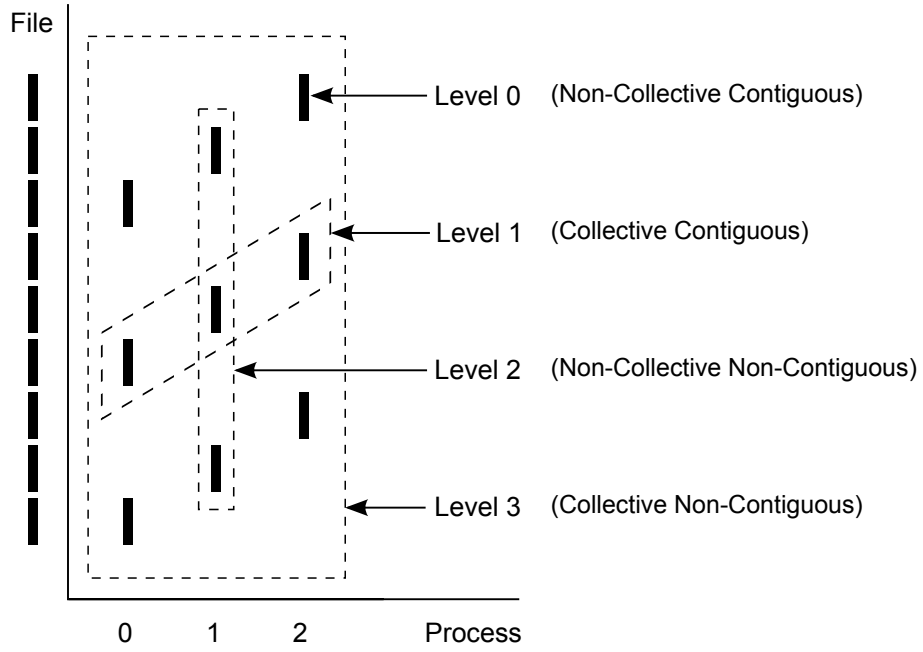
In listing 2.3 we give an example of how to define patterns, each with 10 iterations and $1024 * 1024$ elements per process. This causes that each process gets a 10MB part of the file, since one element has the size of 1 byte. The total file size depends on the number of processes N, participating in the read or write call. In this example the total file size is $N * 10$MB.

Listing 2.3: Parallel I/O Patterns

```
define pattern {"pattern0", 10, (1024 * 1024), 0}
define pattern {"pattern1", 10, (1024 * 1024), 1}
```

Figure 2.2 demonstrates the different levels of access, while a higher level achieves better performance. The framed file-parts describe the way of how the whole file is read/written by multiple processes.

Figure 2.2: Different Levels of Access with 3 Processes



The manual for this benchmark - attached to the source code - provides a detailed explanation of the programming language.

2.3 Implementation

The benchmark has three layers, which are the scanning layer for lexical analysis, the parsing layer for grammatical analysis and the intepreter layer.

2.3.1 Scanner Layer

To build the parser layer we are using Flex and Bison. First off we describe how we create the Flex scanner by giving examples of the actually used lexer in the benchmark. The Flex config file is split in three sections where %% is used as separation mark:

Listing 2.4: Flex Config Structure

```
%{
C definitions
%}
Flex definitions

%%
Rules
%%

User-defined C routines
```

In the definitions section we include necessary header files and define data structures as well as setting Flex options. Grammar rules are defined in the second section and custom C code in the third. Listing 2.5 shows a simple example:

Listing 2.5: Flex Config Snippet

```
%{
#include <stdio.h>
#include <glib.h>
#include "parser.h"
%}

%%
time     return TTIME;
\{       return TEBRACEL;
\}       return TEBRACER;
%%

gchar* strstrip(gchar *string) {...}
```

To define the Flex rules, we use regular expressions as well as simple keywords as shown in listing 2.5. Once Flex recognizes a rule, the code following on the right (usually separated by tab stops) will be executed. Since we want to use a Bison parser on top of Flex, we simply return a keyword token to Bison by writing a return statement. Flex generates the necessary tokens automatically, but we need to define it in Bison manually (Listing 2.6). The custom `strstrip` function removes the leading and trailing quotes (") from strings.

2.3.2 Parser Layer

The Bison config file has the same structure as Flex's. Since Bison does the grammar parsing, we need to define the keyword tokens that Flex returns to the Bison parser. We also need to define grammar rules for each command, where the defined keyword tokens and Bison rules can be used. Listing 2.6 gives an example.

Listing 2.6: Bison Config Snippet

```
%{
#include <stdio.h>
%}
%token TTIME TEBRACEL TEBRACER

%%
commands : /* empty */
         | commands command
         ;

command : time
        ;

time : TTIME TEBRACEL {
           commandList = g_slist_prepend(commandList,
              cmd_new(CMD_TIMESTART, NULL, NULL));
       }
       commands TEBRACER {
           commandList = g_slist_prepend(commandList,
              cmd_new(CMD_TIMESTOP, NULL, NULL));
       }
     ;
%%
```

Due to the fact that the config files of Flex and Bison are similar, we concentrate on the grammar section. The parser needs a root grammar definition which is, by default, the first defined grammar rule. From here on, we build our grammar tree by defining other grammar rules. For the commands rule we use left recursion since this is more efficient then right recursion in Bison [7].

Every time Bison finds a match of a rule, the code between braces {...} is executed. We use this feature to build a parse tree for the interpreter. But since there are still only linear grammar constructs in the benchmark, we decided to compress the tree into a list of commands to allow a less complex implementation.

For example, to implement the time command, we use a start and stop command. So the interpreter knows when to issue and complete a timing measurement. We use a singly linked list from the Gnome Library and prepend the commands on the beginning. This creates an inverse list of commands and needs to be reversed once the source file has been completely parsed. The reason for this is the complexity for appending on `GSList` ($O(n)$) [8].

`cmd_new` creates a new command for the interpreter with the following parameters: the command type, the command name and a pointer to a parameter struct, since some commands need parameters.

Bison can be connected to Flex easily since they are adapted to work together. The entry function of the Flex generated lexer is `yylex()`. Bison's entry function is `yyparse()` which in turn calls `yylex()` each time the next token is requested. To accomplish that Flex passes the recognized tokens to Bison, we always return the token identifier in Flex like in listing 2.5. When we are generating the Bison parser in the make file, it outputs (`-d`) a header file `parser.h` [2] with all necessary definitions which we need to include in the Flex config file. As shown in listing 2.7 we create a case insensitive (`-i`) batch mode (`-B`) Flex scanner with some optimizations (`-CFr`) (see [9]).

Listing 2.7: Generating the Lexer and Parser

```
flex -CFr -i -B -o scanner.c kulga.lex
bison -d -o parser.c kulga.yacc
```

2.3.3 Interpreter Layer

We use several structures from the Gnome Library to implement the interpreting of the generated parse list. For example, we are using `GHashTable` hash tables to implement user defined variables or process groups, `GList` as stacks to realize constructs like loops

[2] The default name `y.tab.c` can be changed with `-o` flag

or timing, and the Better String Library [10] to do the string processing for variable value replacement.

2.4 Outlook & Future Work

There are several points for improvements. The biggest limitation of the todays implementation is the current parse tree structure. It is hold in a list, what allowed us to have a less complex implementation. This especially leads to some restrictions: we cannot efficiently handle conditional branches in the language since we are not able to do forward branches in a list structure. To allow this we need to use a real **abstract syntax tree processing**.

Furthermore, the **language grammar** needs to be improved likewise. This all goes in common with the lack of true **expression processing**. Expressions would basically enable all options for later more complex language constructs, like runtime expression evaluation, functional programming paradigms and similar.

With those three improvements we would create a basis that can be easily extended. That means it would be possible to add things like whole new programming paradigms, conditional language constructs like `if/else` or `while` together with boolean expressions, arithmetic expressions, string concatenation and more.

3 Selected Cluster File Systems

3.1 Test Cluster

The cluster, where we installed the file systems on, has one master which is acting as a gateway to nine compute nodes. Each node is equipped with:

- Two Intel Xeon 2GHz CPUs
- 1GB DDR-RAM
- Four nodes with 80GB IDE HDD
- Five nodes with 160GB SATAII RAID0
- 1GBit Ethernet Interconnection
- Ubuntu "Gutsy Gibbon" 7.10 Server Edition
- MPICH2 1.0.5p4 [11] and GCC 4.2.3 amongst others

3.2 Ceph

3.2.1 Overview

Ceph [12] is a new object based distributed file system, capable of managing many petabytes of storage. It has been developed at the Storage Systems Research Center [13] at the University of California, Santa Cruz. Ceph introduced several innovative concepts, for example CRUSH [14], an algorithm for decentralized placement of replicated data. The main goals are POSIX compliancy, performance and fault tolerance through replication, thus offering no single point of failure [15]. The Ceph system consists of three main components. A POSIX compliant client, a cluster of OSDs representing the

distributed data storage, and a metadata cluster managing the file system's meta data information.

Recently the Ceph developers released several new versions in a relatively short time period. When we started this practical, the installation of Ceph required some manual adaptions in some source files and scripts to make it work properly on our cluster. But with the latest release this adaptions are no longer needed. We describe the installation steps for Ceph 0.14.

3.2.2 Installation

Before installing Ceph, the packages `libboost-dev, libedit-dev` and `libssl-dev` should have been installed. The Ceph source can be downloaded at `http://ceph.newdream.net/download/ceph-0.14.tar.gz` and should be extracted on a place of your choice. We will reference this folder as `<ceph-src>` in the following.

Listing 3.1: Getting Ceph

```
sudo apt-get install libboost-dev libedit-dev libssl-dev

wget http://ceph.newdream.net/download/ceph-0.14.tar.gz
tar xzf ceph-0.14.tar.gz
```

Once extracted, we can build Ceph from sources. Optionally, you can define a custom folder (`--prefix`) where Ceph will be installed. If not set, the default installation target folder will be `/usr`. We assume that `/<ceph-install>` is a shared folder where every node has access to, e.g. via NFS.

Listing 3.2: Compiling Ceph

```
./configure --prefix=/<ceph-install>
make
make install
```

Listing 3.3 is a sample configuration. It defines two nodes, each hosting a monitor (MON), metadata server (MSD) and an object storage server (OSD). The `ceph.conf` file is placed in the `etc` folder within the Ceph installation path `/<ceph-install>`.

Listing 3.3: Ceph Configuration File ceph.conf

```
[global]
        pid file = /var/run/ceph/$name.pid

[mon]
        mon data = /tmp/ceph/mon
[mon0]
        host = node1
        mon addr = 10.0.0.1:6789
[mon1]
        host = node2
        mon addr = 10.0.0.2:6789

[mds]
[mds0]
        host = node1
[mds1]
        host = node2

[osd]
        sudo = true
        osd data = /tmp/ceph/osd
[osd0]
        host = node1
[osd1]
        host = node2

[group everyone]
        addr = 10.0.0.0/24
[mount /]
        allow = %everyone
```

Next we need to create the underlaying file systems for the OSDs on all defined hosts. For our installation, we used the `ext3` file system with user `xattr` support. Initially, we installed the Linux 2.6.30 kernel which is officially supporting the `btrfs` file system, recommended for OSDs. But for some reason, `ext3` performed much better than `btrfs` on meta data operations. Therefore we decided not to use it for Ceph.

Table 3.1 shows the average meta data I/O performance of `ext3` and `btrfs` on the test cluster, determined with the `fileop` tool from the IOzone kit [16]. We only show certain relevant commands measured by `fileop -f 10 -s 1`.

Table 3.1: Average Metadata-Performance of ext3 and btrfs [ops/sec]

FS	mkdir	rmdir	create	close	stat	chmod	link	unlink	delete
ext3	21213	28057	30053	157296	293945	122950	56764	81206	27521
btrfs	5699	4879	5331	119776	266018	22514	4185	7260	4414

To make the OSDs work properly, we need to do some additional cleaning of the created and mounted ext3 partition. As listing 3.4 shows, Ceph data is placed in `/tmp/ceph/` respectively on `sda3` here as an example, but you can choose an arbitrary path and partition.

Listing 3.4: Creating the Storage File System

```
mkdir -p /tmp/ceph/osd
mkdir -p /tmp/ceph/mds

sudo mkfs.ext3 /dev/sda3
sudo mount -t ext3 -o user_xattr /dev/sda3 /tmp/ceph/osd/
sudo rm -vRf /tmp/ceph/osd/*
sudo rm -vR /tmp/ceph/mon/
sudo chmod 777 /tmp/ceph/osd/
```

We create the distributed Ceph file system on all hosts defined in the config file with the `mkcephfs` command:

Listing 3.5: Creating the Ceph File System

```
/<ceph-install>/sbin/mkcephfs --allhosts -v

sudo modprobe libcrc32c
sudo insmod /<ceph-src>/src/kernel/ceph.ko
```

This requires that you can login as root over SSH to all nodes defined. `mkcephfs` is not yet capable to create the Ceph file system in parallel. This means it is necessary to run the script with the `--allhosts` option to get a consistent file system. Additionally, a running Ceph cluster can also be extended afterwards [17, 18].

Together with this step we load the Ceph kernel module, which requires `libcrc32c` to be loaded first. Alternatively there is a Fuse client available, but we had some serious problems using it with Ceph 0.14.

In case of successful installation, Ceph can be started on all hosts (`-a`) with the shipped script like in listing 3.6. In the second step, we mount the distributed file system, by giving the monitor's IP address, to a local mount point `/mnt/ceph`.

Listing 3.6: Starting Ceph

```
/<ceph-install>/etc/init.d/ceph -a start

sudo mkdir /mnt/ceph
sudo mount -t ceph 10.0.0.1:/ /mnt/ceph
```

If root remote login over SSH is not possible, you need to run `/<ceph-install>/etc/init.d/ceph start` on all nodes manually.

3.3 GlusterFS

3.3.1 Overview

GlusterFS [3] is developed by Gluster, formerly known as Z Research. The prototype was delivered in early 2007 and version 2 came out in May of 2009. We tested it at version 2.0.6 and meanwhile 2.0.7 has been released. GlusterFS provides a modular and kernel-independent file system, thus making it capable of scaling to several petabytes and easy to customize and install. It has a client/server-architecture and is set on top of an existing, underlying file system. GlusterFS also provides POSIX-conformity and minimizes dependencies since there is no extra meta-data server. The client uses FUSE [19] to deliver a file system in user space.

Modularity and customizability is achieved by so called translators. They provide a lot of optional features. The following different translator types are available:

- **Performance-Translators** are used, to adjust the filesystem to your workload. Example: `performance/write-behind` aggregates multiple smaller write operations into fewer larger write operations and writes them in background (non-blocking).

- **Protocol-Translators** determine which transport-protocol should be used. Example: `protocol/server` and `protocol/client` implements TCP and Infiniband amongst others.

- **Cluster-Translators** specify how the data should be organized on the servers. Example: `cluster/replicate` stores identic copies on each of the selected server.

- **Encryption-Translators** offer an encrypted transmission of the data between server and client. So far only the sample encryption `encryption/rot-13` is implemented.

- **Feature-Translators** are those translators, which don't fit in the other categories. Example: By using `features/locks` you can get both, advisory locking and mandatory locking support. This also implements more locking mechanisms required for GlusterFS itself.

With these translators it is possible to stack a customized distributed file system. You can specify the appropriate translators in volume-files, which are loaded at the start by server or client.

3.3.2 Installation

GlusterFS is easy to install in single-user environment, since there are pre-built packages for a lot of Unix based operating systems. For example CentOS, Debian, Fedora or Ubuntu. In the multi-user environment, some adjustments have to be done.

Before installing GlusterFS, the packages `flex, bison, byacc` should have been installed. The sources can be downloaded at `http://ftp.gluster.com/pub/gluster/glusterfs/2.0/LATEST/glusterfs-2.0.6.tar.gz` and be extracted on a place of your choice.

Listing 3.7: Getting GlusterFS

```
mkdir gluster
wget http://ftp.gluster.com/pub/gluster/glusterfs/2.0/
   LATEST/glusterfs-2.0.6.tar.gz
tar xzf glusterfs-2.0.6.tar.gz
```

Once extracted, we can build GlusterFS using the default compiler installed.

Listing 3.8: Compiling GlusterFS

```
./configure --prefix=/<gluster-install> --with-
   mountutildir=/<gluster-install>/sbin
make
```

During installation, we faced the problem that `make install` wants to copy the start/stop-script into `/etc/init.d`. But as unprivileged user, we do not have the rights to write in `/etc`. We modified the makefile `/extras/init.d/Makefile` to cause that the init script is placed in the specified custom installation folder.

After replacement of the `initdir` entry `initdir=/etc/init.d` with `initdir=${prefix}/etc/init.d`, we can install GlusterFS as shown in Listing 3.9.

Listing 3.9: Installing GlusterFS

```
make install
```

For better comparability, we use `ext3` as underlying file system. In our sample configuration we use two nodes, a server and a client, where one volume-file per server and client is needed. Using `/mnt/server/` as destination-folder server-side and `/mnt/client/` client-side.

In the `glusterfs-server.vol` volume-file, we define the properties of the server (see listing 3.10). First of all, we set the storage destination as statet in `volume posix`.

The next parts are wrapped around the layers previously defined, indicated by the keyword `subvolumes`. We also add advisory locking and mandatory locking support in `volume locks`. In `volume brick`, we have the performance-translator `performance/io-threads`, to handle more requests at a time. In the last volume, we choose the network protocol (TCP) and the access rights.

Listing 3.10: GlusterFS Volume-File glusterfs-server.vol

```
# file: /<gluster-install>/glusterfs-server.vol

volume posix
  type storage/posix
  option directory /mnt/server
end-volume

volume locks
  type features/locks
  subvolumes posix
end-volume

volume brick
  type performance/io-threads
  option thread-count 2
  subvolumes locks
end-volume

volume server
  type protocol/server
  option transport-type tcp
  option auth.addr.brick.allow *
  subvolumes brick
end-volume
```

The volume-file `glusterfs-client.vol` for the client follows the same concept (see listing 3.11). In `volume remote`, we again define the protocol to use. Additionally we give the address of the server (IP or DNS name) and which volume we want to access on server-side. In `volume writebehind` and `volume cache`, we use the performance translators `performance/write-behind` and `performance/io-cache`. I/O cache helps to reduce the load on the network and the server, when the client is accessing files for reading. Performance improvedments can be expected as long as the files are not modified on the server between reads.

Listing 3.11: GlusterFS Volume-File glusterfs-client.vol

```
# file: /<gluster-install>/glusterfs-client.vol
volume remote
  type protocol/client
  option transport-type tcp/client
  option remote-host 192.168.0.3 # IP address of the
     remote brick
  option remote-subvolume brick
end-volume

volume writebehind
  type performance/write-behind
  option cache-size 1MB
  subvolumes remote
end-volume

volume cache
  type performance/io-cache
  option cache-size 512MB
  subvolumes writebehind
end-volume
```

Starting the Server is the only part where root rights are required. This is because of the low-level I/O control in ClusterFS:

Listing 3.12: Starting GlusterFS Server

```
sudo /<gluster-install>/glusterfsd -f /<gluster-install>/
   glusterfs-server.vol
```

Finally we can start the client:

Listing 3.13: Starting GlusterFS Client

```
/<gluster-install>/sbin/glusterfs -f /<gluster-install>/
   glusterfs-client.vol /mnt/client
```

Alternatively, you can use the init.d script. Because of the previously described install problem, we had to slightly adjust the lines with `SCRIPTNAME` and `CONFIGFILE` in order to use them.

4 Conclusion

We developed a programmable benchmark which is able to test different I/O-patterns defined in an own programming language. These patterns, which can be defined by using sequential as well as parallel I/O commands, can be passed into the benchmark. Even in an advanced stage of implementation, the benchmark has potential for improvements as described above.

Furthermore we learned the ropes of two modern distributed file systems - Ceph and GlusterFS - by installing them on a test cluster. We presented its installation and some challenges, which we faced to. Ceph is not yet capable for productive usage, but GlusterFS is already used in productive environments by several institutions [20].

List of Figures

List of Tables

Listings

Bibliography

[1] Flex. http://flex.sourceforge.net, October 2009.

[2] Bison. http://www.gnu.org/software/bison/, October 2009.

[3] GlusterFS. http://www.gluster.com, October 2009.

[4] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. http://www.usenix.org/events/osdi06/tech/weil.html, November 2006.

[5] The Lex & Yacc Page. http://dinosaur.compilertools.net, October 2009.

[6] Gnome Library. http://library.gnome.org, October 2009.

[7] Bison Manual: Recursive Rules. http://www.gnu.org/software/bison/manual/bison.html#Recursion, October 2009.

[8] Gnome Library Reference Manual: Singly-Linked Lists. http://library.gnome.org/devel/glib/stable/glib-Singly-Linked-Lists.html#g-list-append, October 2009.

[9] Flex Manual: Options for Scanner Speed and Size. http://flex.sourceforge.net/manual/Options-for-Scanner-Speed-and-Size.html#Options-for-Scanner-Speed-and-Size, October 2009.

[10] Paul Hsieh. The Better String Library. http://bstring.sourceforge.net, October 2009.

[11] Argonne National Laboratory. MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/, October 2009.

[12] Ceph. http://ceph.newdream.net/, October 2009.

[13] Storage Systems Research Center. Petabyte Scale Object-Based Storage Systems @ SSRC. http://www.ssrc.ucsc.edu/proj/ceph.htm, May 2008.

[14] Sage Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. `http://www.ssrc.ucsc.edu/Papers/weil-sc06.pdf`, November 2006.

[15] Ceph Wikipedia Article. `http://en.wikipedia.org/wiki/Ceph`, October 2009.

[16] IOzone. `http://www.iozone.org`, October 2009.

[17] Ceph: OSD cluster expansion/contraction. `http://ceph.newdream.net/wiki/OSD_cluster_expansion/contraction`, October 2009.

[18] Ceph: Monitor cluster expansion. `http://ceph.newdream.net/wiki/Monitor_cluster_expansion`, October 2009.

[19] FUSE, Filesystem in Userspace. `http://fuse.sourceforge.net`, October 2009.

[20] Who is using GlusterFS. `http://gluster.com/community/documentation/index.php/Who%27s_using_GlusterFS`, October 2009.