Universität Hamburg

Fachbereich Informatik

Bachelorarbeit

Replay Engine for Application Specific Workloads

Name:	Jörn Ahlers
Matrikelnummer:	6053193
Betreuer:	Julian Kunkel
Abgabe Datum:	12.04.2012

Ich versiche, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 12. April 2012

Jörn Ahlers

Acknowledgments

First i want to thank Julian Kunkel for the time to guide and support me during this bachelor thesis.

I also want to thank my parents for all the support they gave me.

Abstract

Today many tools exist which are related to the processing of workloads. All of these have their specific area where they are used. Despite their differences they also have functions regarding the creation and execution of workloads in common. To create a new tool it is always needed to implement all of these functions even when they were implemented before in another tool.

In this thesis a framework is designed and implemented that allows replaying of application specific workloads. This gets realized through a modular system which allows to use existing modules in the creation of new tools to reduce development work. Additionally a function is designed to generate parts of the modules by their function headers to further reduce this work. To improve the generation, semantical information can be added through comments to add advanced behavior.

To see that this approach is working examples are given which show the functionality and evaluate the overhead created through the software.

Finally additional work that can be done to further improve this tool is shown.

Contents

1 Introduction

This chapter provides information to help understanding of the topic. It states the problem which is worked on by this thesis and shows related works to this topic. Furthermore, it states the goals and gives an outlook to the following chapters.

1.1 Problem Statement

There are many programs related to workload processing which have specific application domains. One area is benchmarking [1][5]. Due to the complexity of computer systems it is hard to compare the computing power between systems by just looking at the specification. Instead, it is mandatory to determine performance with real applications. Benchmarks are programs that measure the performance of a given system. Benchmarks can be done for whole computer systems or individual components. To ensure that results are comparable, a test has to generate results with a low variation between multiple runs and every system has to run the same test. With such a standardized benchmark it is then possible to compare system performance or to create a ranking like the TOP500.[11][3]

Benchmarks are often designed to create a synthetic workload that stress specific aspects of a computer system. There are many workloads imaginable, each result in different behavior. Usually, the results of a synthetic benchmark can not predict performance of a real application, because application workloads are much more complex. For example, a HDD is much faster for sequential access than for random access. Therefore, when deciding on the best hardware for a given software by benchmark results the way how the results were obtained has to be considered. Reasons in doing benchmarks instead of running the actual program are that a program can not be ported to the other system easily, or that its license might restrict usage. Benchmarks are also preferred when the application is complex with dependencies to other libraries, which make it hard and time consuming to set up where the setup of a benchmark is much faster. In this case benchmarking helps to predict how well the application will run.

Example tools for benchmarking I/O are IOZone[16] and Parabench[13]. IOZone is capable to execute recorded I/O patterns. This enables to mimic applications I/O behavior. Parabench is a tool for programmable parallel I/O benchmarks. This allows the creation of benchmarks that match application behavior.

Usually, in order to optimize program behavior, activity of the program are recorded in so-called trace files for analysis. Traces are like log files from a program run; they contain information about the executed functions at program runtime, the parameters that were used as well as the returned values. This information can be visualized and analyzed by a developer, which allows identification of bottlenecks and communication errors. Depending on the settings more or less information can be stored. Trace files can be used to find undesired behavior.

Example tools for tracing are EZTrace[2] and VampirTrace[12][8]. They are capable of tracing programs with the commonly used interfaces like Message Passing Interface (MPI)[10], Open Multi-Processing (OpenMP)[4], Portable Operating System Interface Thread (POSIX Thread, PThread)[14] and Performance Application Programming Interface (PAPI)[9]. To support third-party interfaces they offer a wrapper to generate plugins which support that interface.

The idea is to create a generic framework that allows processing of workloads in different ways like doing benchmarks, replaying the workload, simulation of workloads and the conversion in different file formats.

All these tasks have parts in common during creation and execution of workload: timing of operations, errors must be handled to check correctness, maybe the user wants to get some statistical information. This functionality is re-implemented in each benchmark replay engine. However, all existing tools are limited to specific domains. The aim of this thesis is to create an universal framework that supports the common processes through a modular system. With the help of the modules it is then possible to reuse functionality instead of rewriting it for every benchmark. As the execution of each function is very similar but dependent on its parameter types, this process should be mostly automated.

1.2 Related work

In this section an excerpt of existing tools and solutions to problems related to trace generation and benchmarking is given.

1.2.1 IOzone

IOZone is a tool for synthetic file system benchmarks. It supports different read and write access patterns in order to measure file system (and block device) behavior in different read and write situations. Some of the possible pattern are:

- **Read:** A sequential read of an already existing file.
- **Write:** A sequential write of a new file on the storage. Additionally to the file data the storage drive also needs to write metadata of the storage file system.
- **Re-Read / Re-Write:** An additional read / write after the previous one. This might be faster as the file could be still in the cache and metadata is already written.
- **Random Read / Write:** Reads and Writes occur randomly. Performance might be lower because of additional overhead of the storage technology, for example, due to seek times in a hard disk drive.
- **Fread / Fwrite:** Testing of read and write through the `fread()` and `fwrite()` library functions.

IOZone measures either the bandwidth or the latency of the operations. The results are printed to the standard output or they can be written to an Excel file for later analysis.

IOZone also offers a functionality to use a specific file behavior. This is used to benchmark a file system with a known behavior from an application. This way it is possible to get an estimate for performance of a real application. IOZone realizes this with a file that contains the pattern of the operations: Each operation consists of three values which represent the start position in the file, the length in bytes to read and a wait time in milliseconds until the next operation starts.

1.2.2 EZTrace

EZTrace[2] is a performance analysis tool for Linux and Mac. To visualize the generated traces another program (like ViTE[1]) is needed as EZTrace does only the tracing. Supported programming models (and interfaces)

[1]Visual Trace Explorer `http://vite.gforge.inria.fr/`

are PThread, MPI, OpenMP, PAPI, standard I/O and memory allocation. The structure is plugin based: Every programming model is a own plugin to allow the addition of additional models.

To trace with EZTrace the first step is to select the interfaces that should be traced. After the configuration is finished the tracing can start and for each process a own trace file is written. These trace files can then be merged into one trace file. In this process filters can be applied to restrict the generated traces to certain program models. It is also possible to compute statistics from a trace file.

For creating additional plugins EZTrace offers a wrapper generator. With this it is possible to generate plugins for tracing from the function headers. The generated plugin can then be used to trace the specified interface.

To illustrate the creation of a new wrapper a small example is generated. For the generation of a new plugin a file with the name of the traced library, a short description, a plugin ID and the headers of the functions that are traced is needed. Listing 1.1 shows a header file named `example.hdr` which will be used as an example here.

```
BEGIN_MODULE
NAME arithmetic
DESC "arithmetic functions"
ID a0
int* plus(int* num1, int* num2)
int* minus(int* num1, int* num2)
END_MODULE
```

Listing 1.1: Example creation of a new plugin for the plus and minus functions

With the `eztrace_create_plugin` tool the source code and a makefile is generated from `example.hdr`. The created source codes are shown in Listing 1.2 and Listing 1.3. Then the new plugin can be compiled and used with EZTrace to intercept the specified functions.

```
#include "eztrace.h"
#include "arithmetic_ev_codes.h"

int* (*libplus) (int* num1, int* num2);

int* (*libminus) (int* num1, int* num2);

int* plus (int* num1, int* num2) {
        EZTRACE_EVENT2 (FUT_arithmetic_plus_1 , num1, num2);
        int* ret = libplus (num1, num2);
        EZTRACE_EVENT2 (FUT_arithmetic_plus_2 , num1, num2);
        return ret;
}

int* minus (int* num1, int* num2) {
        EZTRACE_EVENT2 (FUT_arithmetic_minus_3 , num1, num2);
        int* ret = libminus (num1, num2);
        EZTRACE_EVENT2 (FUT_arithmetic_minus_4 , num1, num2);
        return ret;
}

void __arithmetic_init (void) __attribute__ ((constructor));
/* Initialize the current library */
void
__arithmetic_init (void)
{
  INTERCEPT("plus", libplus);

INTERCEPT("minus", libminus);
```

```
  /* start event recording */
#ifdef EZTRACE_AUTOSTART
  eztrace_start ();
#endif
}

void __arithmetic_conclude (void) __attribute__ ((destructor));
void
__arithmetic_conclude (void)
{
  /* stop event recording */
  eztrace_stop ();
}
```

Listing 1.2: Code in arithmetic.c generated by `eztrace_create_plugin` with `example.hdr`

```
#ifndef __arithmetic_EV_CODES_H__
#define __arithmetic_EV_CODES_H__

/* This file defines the event codes that are used by the example
 * module.
 */
#include "ev_codes.h"

/* Event codes prefix. This identifies the module and thus should be
 * unique.
 * The 0x0? prefix is reserved for eztrace internal use. Thus you can
 * use any prefix between 0x10 and 0xff.
 */
#define arithmetic_EVENTS_ID     0xa0
#define arithmetic_PREFIX        (arithmetic_EVENTS_ID << NB_BITS_EVENTS)

/* Define various event codes used by the example module
 * The 2 most significant bytes should correspond to the module id,
 * as below:
 */
#define FUT_arithmetic_plus_1 (arithmetic_PREFIX | 0x1)
#define FUT_arithmetic_plus_2 (arithmetic_PREFIX | 0x2)
#define FUT_arithmetic_minus_3 (arithmetic_PREFIX | 0x3)
#define FUT_arithmetic_minus_4 (arithmetic_PREFIX | 0x4)

#endif /* __arithmetic_EV_CODES_H__ */
```

Listing 1.3: Code in `arithmetic_ev_codes.h` generated by `eztrace_create_plugin` with `example.hdr`

1.2.3 Parabench

Parabench is a programmable I/O benchmarking tool that allows evaluating the file system with different I/O behavior. The goal is to mimic I/O behavior of arbitrary applications as close as possible, because every application has its own requirements regarding I/O. The closer the benchmark test is to the real application needs, the more accurate and meaningful the result predicts application performance on the given system. Parabench itself is open source and uses a modular architecture. This allows the extension and adjustments to personal needs. To create the benchmark patterns parabench uses its own programming language. As the benchmark can be feed with arbitrary patterns, it is possible to adapt it without creating a complete new benchmark. Currently, this language provides support for MPI and parallel I/O.

The example in Listing 1.4 uses two patterns that are used for reading and writing of 100 MB of data per process. This is done in 102400 reads/write with a size of 1 KB each and 10 reads/writes with a size of 10 MB each. Parabench was run with this test program on a cluster using 16 processes each reading and writing 100 MB for both patterns. After completion parabench outputs the rank of the process and the time spent to write the 100 MB of data.

```
// Define spatial access patterns for parallel access.
// Access 1 KiB of data 102,400 times.
define pattern {"102400*1k-lvl0", 102400 , (1 * 1024) , 0};
define pattern {"10*10m-lvl0", 10, (10 * 1024 * 1024) , 0};

$p = "pvfs2 :// pvfs2 ";
// Execute a barrier to synchronize all processes.
barrier;
// Parallel write for pattern "102400*1k-lvl0" and time the duration.
time [" Write 102400*1k L0"] pwrite ("$p/lvl0", "102400*1k-lvl0 ");

barrier;
// Parallel read for pattern "102400*1k-lvl0"
time [" Read 102400*1k L0"] pread ("$p/lvl0", "102400*1k-lvl0 ");

barrier;
// Parallel write for pattern "10*10m-lvl0"
time [" Write 10*10m L0"] pwrite ("$p/lvl0", "10*10m-lvl0 ");

barrier;
// Parallel read for pattern "10*10m-lvl0"
time [" Read 10*10m L0"] pread ("$p/lvl0", "10*10m-lvl0 ");
```

Listing 1.4: Synthetic MPI-IO test program in parachbenchs langugage

1.2.4 VampirTrace

VampirTrace is a tool that records application behavior for post-mortem performance analysis. It consists of a library for software instrumentation and a tool set for the visualization. The instrumentation part generates traces of software runs which can then analyzed by the visualization tool set. Vampir supports MPI, Pthread, OpenMP, PAPI, Java and third-party library tracing. The third-party library tracing is realized by a experimental wrapper that generates the source for a plugin which can then included into tracing. Traces are visualized with a timeline[2] for every thread. This enables the user to track the behavior of a given program by assessing communication between threads and to check if it works as intended. VampirTrace is furthermore able to save the traces to the Open Trace Format (OTF)[7] which allows the processing with 3rd party tools. OTF is an open source format developed for applications in the high performance sector. It was developed to offer a open, flexible and efficient way to work with trace data. The flexibility is achieved by the representation of the data which can be adjusted to the needs of the application by setting parameters.

1.3 Goal of the Thesis

Main goal of this thesis is to create a universal framework that simplifies the task of generating replay engines. Therefore, it should combine common capabilities of all the mentioned tools in a novel framework and it should provide additional capabilities:

[2]Events are displayed in chronological order

- Modularity: It should be possible to provide plugins for replaying additional interfaces without changing existing modules. Also it must be possible to write plugins that mimic existing behavior (interpretation of the parabench language, IOZone input format).
- Performance: The framework must provide an efficient processing scheme to allow execution of activity without stalling.
- Simplify plugin creation: Writing a plugin for additional interfaces involves repetitive tasks that should be reduced as much as possible, for example, by providing templates.

With these three attributes the framework should aim to replace existing replay mechanisms that are implemented across various tools.

1.4 Structure of the Thesis

In Chapter 2 design decisions are discussed that where made during the creation of the new *Tracereplay* framework. Chapter 3 shows how to execute Tracereplay. Additionally details about the output are shown. Chapter 4 handles the creation of new plugins. It shows the steps that are needed to create new plugins. Chapter 5 evaluates Tracereplay with the help of examples. Therewith, the code generation and the performance is assessed. Chapter 6 summarizes and concludes this thesis.

2 Design

This chapter shows how Tracereplay is designed. First an overview is given about the contents of this chapter. Each part is then described in more detail in the respective sections. The sections are: Plugins, where the functionality of each plugin is described. Processing by Tracereplay, which shows how Tracereplay works and how it manages the plugins and their data. Automatic Generation of Plugins, which introduces a script to reduce work on the plugin creation.

2.1 Overview

To meet the design goal for modularity, the program is designed to be plugin based. The reason for the plugin based design is to reach a loose coupling.

There are different needs that each plugin has to fulfill. There have to be ways to read data from files and to write or execute them. As this two tasks are much different there are two types of plugins for them: input and output. An additional filter plugin allows to restrict processed commands. For example to only execute parts of the available commands. Plugins are described in more detail in section 2.2.

The program consists of two parts: controller and processor. The main part is the controller. The controller is responsible for the starting phase. This includes the command line argument parsing and preparation of the plugins. As well as the data transfer between plugins.

The processor is the part that receives data from the controller and which initiates necessary steps in filtering and execution. Additionally, the processor collects statistical data about the run, and displays it when the program completes. Section 2.3 shows how this gets realized.

As structure and parts of plugins repeat often, templates and a generation script are needed. Templates offer the basic structure of plugins and the script allows to generate parts of plugins to reduce the manual work for the creation. The generation is done with informations about the functions. This is shown in Section 2.4.

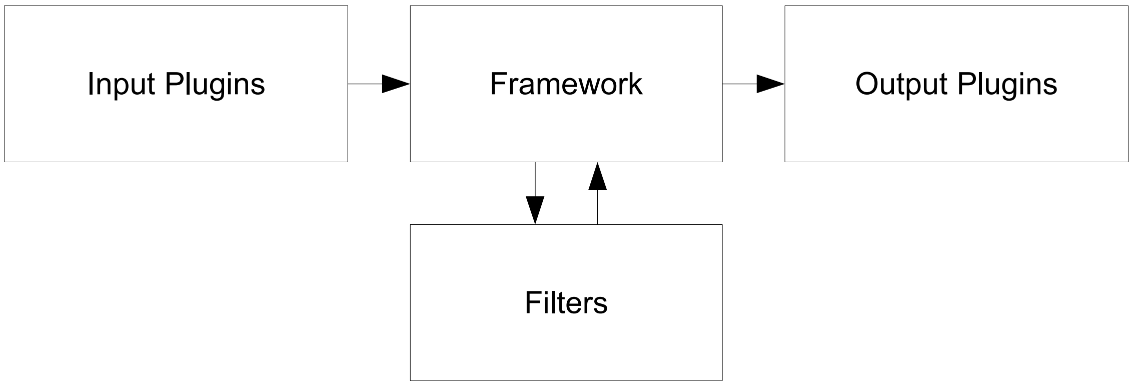

Figure 2.1: Data flow between plugins and framework

Figure 2.1 shows how the data is handled. The input plugins create data which is given to the framework. The data is then passed to the filter plugins and back to the framework. The filtered data is then given to the output plugins where it gets executed.

Data is transferred as so-called *commands* which are C structures that are generated in the input plugin. The decision to use structs was made to keep the data belonging to one function together. Structs also allow passing of a single pointer (to the struct) to functions, instead of passing all the parameters. The struct pointer makes it possible to offer functions that take all the structs. That way the main program does not have to be changed to support arbitrary or even new plugins. With parameter passing a function with variable parameter count would be needed and some way to determine the parameter types.

Commands need to have at least information about the interface they belong to and the name of the function. These information are required by Tracereplay to successfully process a command.

2.2 Plugins

There are parts that all plugins have in common. Before plugins are ready to be used they have to get initialized. Maybe a plugin must prepare itself by setting up variables, create tables and so on. Another functionality they have in common is the ability to receive options. They are used to specify the working data and to customize the behavior.

2.2.1 Input Plugins

Input plugins create commands with are then processed. Every time a new command is needed from the framework it invokes a function on the input plugin. The plugin then delivers the next command. This procedure is repeated until the plugin delivers no more commands. How the commands are generated is up to the plugin. It is possible to have more than one input plugin, but obviously at least one is needed for a successful run. If more than one is loaded they are then handled parallel.

2.2.2 Filter Plugins

Filter plugins are responsible for filtering unwanted commands. Every command created by an input plugin is passed to the filters if any filter is active. As every command belongs to a specific interface only filters that support that interface are used for filtering. Filters that do not support the required interface are skipped.

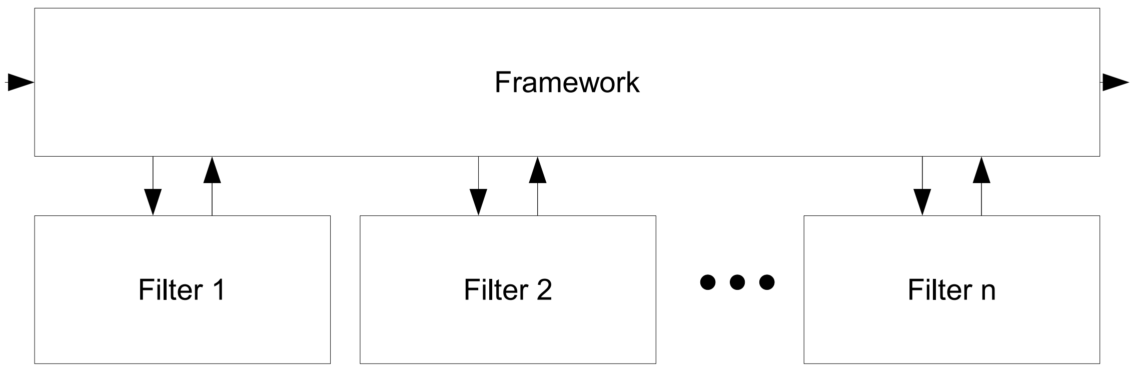

Figure 2.2: Chaining of filter plugins

If more than one filter is running, the filters will be chained and the command is processed by all filters that support the interface – this process is illustrated in Figure 2.2. The command is created by an input plugin and is then passed by the framework to the first filter that supports the command. After the filtering is done the result is returned to the framework which then looks for the next supporting filter. This procedure is repeated until all filters are applied or one filter rejects the command.

The decision inside a filter plugin can be made based on the values of the command, statistical data collected over the runtime or by external data read by the plugin.

2.2.3 Output Plugin

Output plugins process commands, that means they perform some operation driven by the command. For example, by executing the function that is described in the command struct. For every command a separate function is called in the plugin. The functions determines how the command is handled like a execution of the command function or a write to a text file. To check for errors every function returns an integer. Success is indicated with zero, that means if this value is zero everything went as it should. If a value different than zero is returned, this will be interpreted as an error. In this case a warning with the returned value is given with details about the command including all values of the struct.

2.3 Processing by Tracereplay

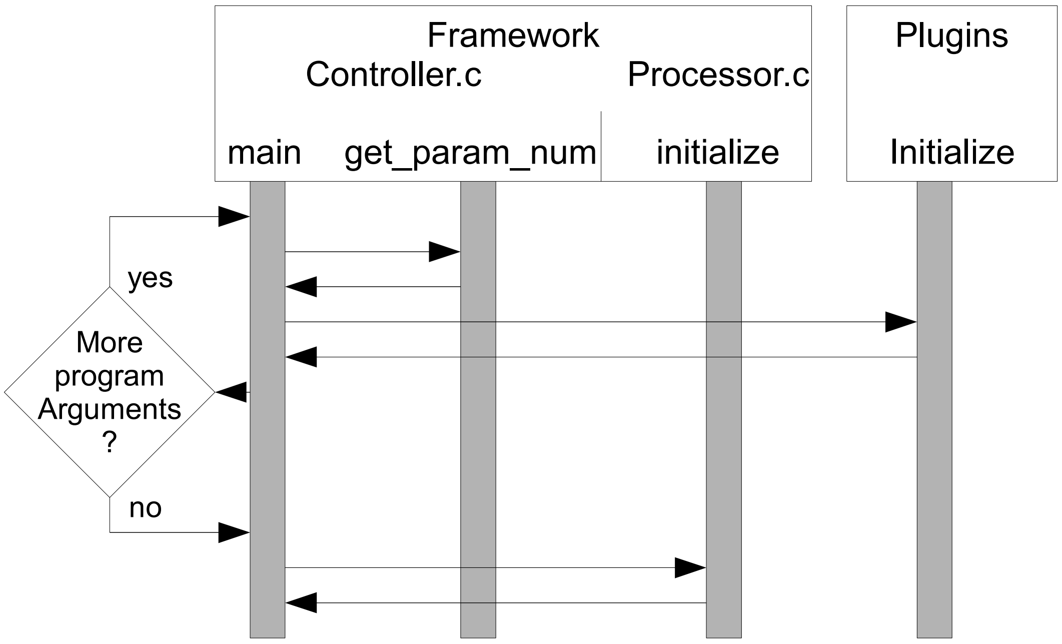

Figure 2.3: Diagram of the program start phase

Figure 2.3 shows the internal processing of Tracereplay.

Parameter handling It starts at the main function where program parameters are parsed. The first parameter is then checked by `get_param_num()`. Program internal parameters have to be given first and are now processed. Invalid parameters produce a warning to inform the user. When a parameter for a plugin is found, the plugin gets loaded. Any program parameters that does not load a plugin is now treated as additional parameter for the previously loaded plugin; these parameters are collected in a list. This list is then passed to the initialize function to prepare the plugin. After the options are handled the plugin is added to a list of active plugins. If more program parameters are available the cycle starts again until no more parameters are left. The last steps in the starting phase are to initialize the processor with the generated plugin list and to create a thread for every loaded plugin.

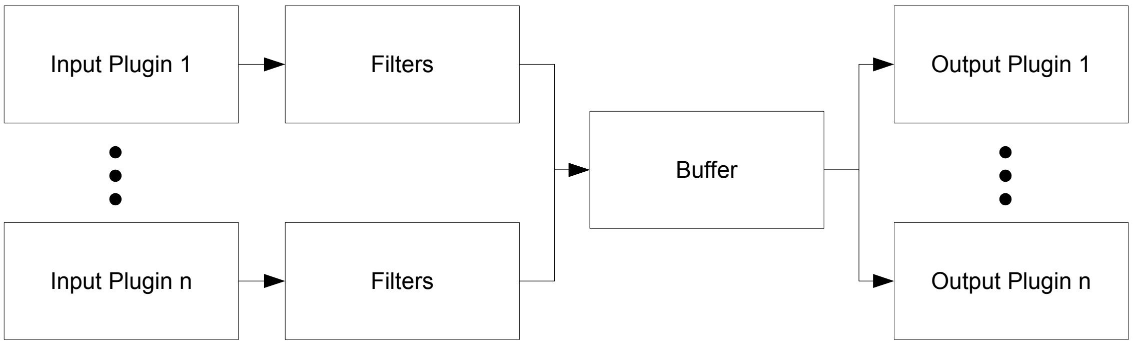

Figure 2.4: Parallel processing of plugins

Threaded execution To ensure that processing of input plugins and filters does not reduce the speed at which operations are processed Tracereplay is parallelized: Every input plugin has its own thread which fetches the commands and executes the filters. Additionally a buffer is created to prevent idle times due to varying command execution times. Figure 2.4 shows the Input plugins where the thread gets a command. This command is then filtered through the filter chain and written to the buffer. To prevent errors through parallel changes on the buffer synchronization is added which allows one thread to modify at a time.

The buffered commands are read by another thread and distributed to all output plugins. This is done by only one thread to prevent the output plugins from racing each other [15]. If the execution of the output is time intense, the execution should be done by a plugin intern thread. This thread is then used solely for the distribution of commands. The parallelization of plugins is up to their needs. Every plugin can create as many threads as it needs for running.

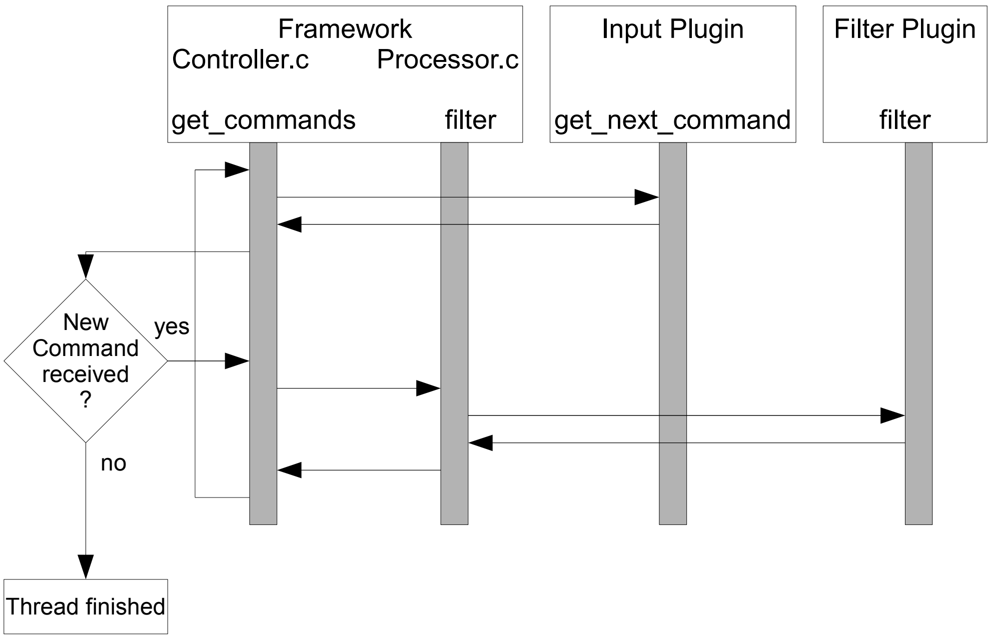

Figure 2.5: Processing of the input data with threads

Figure 2.5 shows how threads handle the input plugins. First, the thread polls the input plugin for the next command. This can either be a pointer to a command struct or a null pointer. A null pointer indicates to the controller the plugin is finished and it is removed from the active plugins. The thread is then no longer needed and terminates. In case of a pointer to a command the struct is given to the filtering in the processor. Here the interface of the command is checked with the interface of the loaded filters. When they match the command is filtered by the plugin. The returned command pointer can now be null when it was filtered out. If that happens the filtering is finished and null is returned to the controller which skips writing to the buffer. As long as the filters do not return null the filtering continues until all filters are used. When a command is returned and there is free space in the buffer writing rights are acquired to append the command to the buffer. Otherwise the thread is suspended until an element in the buffer gets consumed [6]. After a write completes, the next command is read.

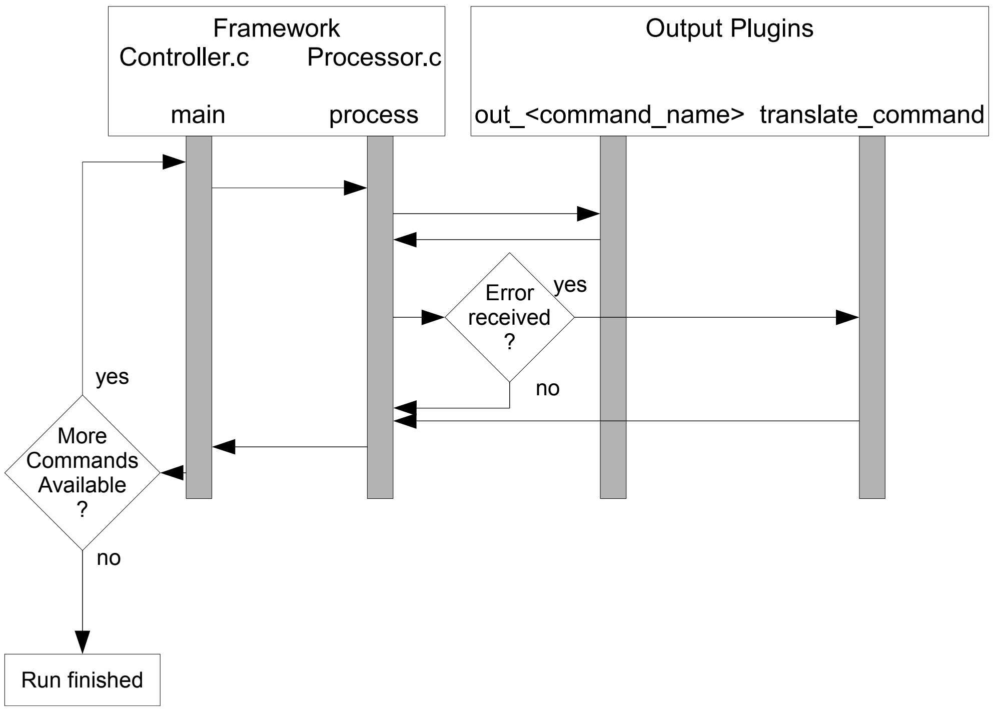

Figure 2.6: Diagram of the threads for the output plugins

Figure 2.6 shows how the thread that handles the output plugins works. The cycle begins with acquiring read rights on the buffer to get a element from the buffer. The command is given to the processor. There the supported interfaces of the output plugins get compared with the interface of the command. When they match the function corresponding to the command name gets called. This function returns an error code. If this is zero everything went fine and the next output plugin is checked. If the value is not zero the output plugin indicates an error. The command is then passed to the plugins `translate_command()` function to get the name and values from the command that produced the error. This is needed because the framework does not know the structure of the command. From the command a warning is generated that displays the command name, the error number and the parameters which were used. Then the next output plugin is handled. When all output plugins were called the next command is acquired from the buffer. This cycle continues until no more elements are in the buffer and all input plugins are finished. The last steps are to print the statistics of the run and to clean up reserved variables. The statistics show the number of executed commands, the number of filtered out commands and how many of them returned an error. For error messages that can occur during runtime see appendix A.1.

2.4 Automatic Generation of Plugins

Some code is repeated in the output plugins and in the command creation. For example, in a replay engine the real function must be executed with the parameters that are provided in the command struct; this functionality is usually replicated. To reduce the work an automatic generation script is designed and developed. As code related to output plugins and command creation depends on the header of the function, it is possible to create a wrapper script that generates code from the header file. It is possible to create the entire function for the struct from the header as function name, parameter names and parameter types can be found in the header file. For the output plugin it is possible to generate the output functions. Since the header file does not contain semantical information, advanced functionality such as error checking or mapping of pointers can not be created from a header file alone. Mapping of pointers is needed because input plugins do not know the real pointers created in output plugins. So input plugins specify a value as a label for the created pointer in the output plugin. Now every time this label is used in a function where a pointer is expected it has to be translated to the actual pointer.

2.4.1 Command Creation

To make the command creation and the reading of parameters in case of an error easier, a helper class is created which is shown in Listing 2.1. To function it first needs its hashtables initialized and an interface specified that the created commands will have. This is done by calling `create_hashtable` with the name of the interface.

When `create_command()` is called with the command name and the commands parameters, a new struct is created. Now information about the parameters which have to be written to the struct are needed. These are supplied by the previous created hashtable. With this information `writeUnion` is called. This function uses the information about the datatype to retrieve the values and to write them to the struct.

The `get_command_string` function allows to create the string for a potential error automatically. For this procedure the hashtable with the parameter names is needed as the function has no knowledge about the meaning of the parameters inside the struct. Together with the parameter type table a string is build from the parameters and returned.

```
#include "commands.h"

GHashTable * command_table;
GHashTable * name_table;
va_list params;
char * command_interface;

//Create hashtables for command creation and set interface for the created commands
void create_hashtable(char *interface) {
  command_interface = interface;
  command_table = g_hash_table_new(g_str_hash, g_str_equal);
  name_table = g_hash_table_new(g_str_hash, g_str_equal);
  g_hash_table_insert(command_table, "open", "\x2\x3\x4\x3");
  g_hash_table_insert(command_table, "creat", "\x2\x4\x3");
  g_hash_table_insert(command_table, "close", "\x3\x3");
  g_hash_table_insert(command_table, "write", "\x3\x1\x6\x5");
  g_hash_table_insert(command_table, "read", "\x3\x1\x6\x5");
  g_hash_table_insert(command_table, "pread", "\x3\x1\x6\x7\x5");
  g_hash_table_insert(command_table, "pwrite", "\x3\x1\x6\x7\x5");
  g_hash_table_insert(command_table, "lseek", "\x3\x7\x3\x7");
  g_hash_table_insert(command_table, "fopen", "\x2\x2\x1");
  g_hash_table_insert(command_table, "fdopen", "\x3\x2\x1");
  g_hash_table_insert(command_table, "freopen", "\x2\x2\x1\x1");
  g_hash_table_insert(command_table, "fclose", "\x1\x3");
  g_hash_table_insert(command_table, "fread", "\x1\x6\x6\x1\x6");
  g_hash_table_insert(command_table, "fwrite", "\x1\x6\x6\x1\x6");
  g_hash_table_insert(command_table, "fseeko", "\x1\x7\x3\x3");
  g_hash_table_insert(command_table, "fseek", "\x1\x8\x3\x3");
  g_hash_table_insert(command_table, "fsetpos", "\x1\x1\x3");
  g_hash_table_insert(name_table, "open", "pathname\0flags\0mode\0return_val");
  g_hash_table_insert(name_table, "creat", "pathname\0mode\0return_val");
  g_hash_table_insert(name_table, "close", "fd\0return_val");
  g_hash_table_insert(name_table, "write", "fd\0buf\0count\0return_val");
  g_hash_table_insert(name_table, "read", "fd\0buf\0count\0return_val");
  g_hash_table_insert(name_table, "pread", "fd\0buf\0count\0offset\0return_val");
  g_hash_table_insert(name_table, "pwrite", "fd\0buf\0count\0offset\0return_val");
  g_hash_table_insert(name_table, "lseek", "fd\0offset\0whence\0return_val");
  g_hash_table_insert(name_table, "fopen", "path\0mode\0return_val");
  g_hash_table_insert(name_table, "fdopen", "fd\0mode\0return_val");
  g_hash_table_insert(name_table, "freopen", "path\0mode\0stream\0return_val");
  g_hash_table_insert(name_table, "fclose", "fp\0return_val");
  g_hash_table_insert(name_table, "fread", "ptr\0size\0nmemb\0stream\0return_val");
  g_hash_table_insert(name_table, "fwrite", "ptr\0size\0nmemb\0stream\0return_val");
  g_hash_table_insert(name_table, "fseeko", "stream\0offset\0whence\0return_val");
  g_hash_table_insert(name_table, "fseek", "stream\0offset\0whence\0return_val");
  g_hash_table_insert(name_table, "fsetpos", "stream\0pos\0return_val");
}

//Create a new command from the parameters
struct Command * create_command(char *function, ...)
{
  struct Command* newCommand = calloc(1,sizeof(struct Command));
  newCommand->type = function;
  newCommand->interface = command_interface;
  va_start(params, function);
  char * types = g_hash_table_lookup (command_table,function);
  if (types == NULL) {
    g_warning("Unknown function used in command creation: %s",function);
  }
  else {
    writeUnion(&(newCommand->u),types);
  }
  return newCommand;
}

//Write parameters to the command struct with information about parameter types
void writeUnion(void * write_pos,char* param_types) {
  int type_pos = 0;
  while(param_types[type_pos] != '\0') {
    switch((int)param_types[type_pos]) {
      case 1:
        *((void**)write_pos) = va_arg(params, void*);
        write_pos += sizeof(void*);
      break;
      case 2:
        *((char**)write_pos) = va_arg(params, char*);
        write_pos += sizeof(char*);
      break;
      case 3:
        *((int*)write_pos) = va_arg(params, int);
        write_pos += sizeof(int);
      break;
      case 4:
        *((mode_t*)write_pos) = va_arg(params, mode_t);
        write_pos += sizeof(mode_t);
      break;
      case 5:
        *((ssize_t*)write_pos) = va_arg(params, ssize_t);
        write_pos += sizeof(ssize_t);
      break;
      case 6:
        *((size_t*)write_pos) = va_arg(params, size_t);
        write_pos += sizeof(size_t);
      break;
      case 7:
        *((off_t*)write_pos) = va_arg(params, off_t);
        write_pos += sizeof(off_t);
      break;
      case 8:
        *((long*)write_pos) = va_arg(params, long);
        write_pos += sizeof(long);
      break;
      default:
        g_error("Illegal Datatype used in struct creation: %d", param_types[type_pos]);
    }
    type_pos++;
  }
}

//Translate command into a string
void get_command_string(struct Command * s_command, char * buf) {
  void * read_pos = &(s_command->u);
  char * types = g_hash_table_lookup (command_table,s_command->type);
  char * names = g_hash_table_lookup (name_table,s_command->type);
  int type_pos = 0;
  char tmp[256];
  while(types[type_pos] != '\0') {
    switch(types[type_pos]) {
      case 1:
        sprintf(tmp,"%s: %p, ",names,*(void**)read_pos);
        strcat(buf,tmp);
        read_pos += sizeof(void*);
      break;
      case 2:
        sprintf(tmp,"%s: %s, ",names,(char*)read_pos);
        strcat(buf,tmp);
        read_pos += sizeof(char*);
      break;
      case 3:
        sprintf(tmp,"%s: %lld, ",names,(long long int) *(int*)read_pos);
        strcat(buf,tmp);
        read_pos += sizeof(int);
      break;
      case 4:
        sprintf(tmp,"%s: %lld, ",names,(long long int) *(mode_t*)read_pos);
        strcat(buf,tmp);
        read_pos += sizeof(mode_t);
      break;
      case 5:
        sprintf(tmp,"%s: %lld, ",names,(long long int) *(ssize_t*)read_pos);
        strcat(buf,tmp);
        read_pos += sizeof(ssize_t);
      break;
      case 6:
        sprintf(tmp,"%s: %lld, ",names,(long long int) *(size_t*)read_pos);
        strcat(buf,tmp);
        read_pos += sizeof(size_t);
      break;
      case 7:
        sprintf(tmp,"%s: %lld, ",names,(long long int) *(off_t*)read_pos);
        strcat(buf,tmp);
        read_pos += sizeof(off_t);
      break;
      case 8:
        sprintf(tmp,"%s: %lld, ",names,(long long int) *(long*)read_pos);
        strcat(buf,tmp);
        read_pos += sizeof(long);
      break;
      default:
        g_error("Illegal Datatype used while reading struct");
    }
    type_pos++;
    if(types[type_pos] != '\0') {
      while (*names != '\0') {
        names++;
      }
      names++;
    }
  }
}
```

Listing 2.1: Helper functions for the struct handling with POSIX IO functions as an example

2.4.2 Annotations

To allow an automatic generation with semantical information an annotation system has been developed: Through comments in the header file information can be added for functions. This information is read by the wrapper to generate additional code. To meet the demand for arbitrary replay engines, comments should be configurable. This gets realized through a configuration file that contains comment names and the code that gets included in the generated source code. As different programming interfaces have different needs, it is possible to create configurations for different interfaces. These configurations can then be imported to be used with the standard configuration. To specify where the code is added to the source file, the configuration file contains different lists which represent the place to add.

Figure 2.7 shows how the calling of the wrapper works. The generation starts with collecting of the functions from the header file. From these functions then the names, types and annotations are extracted and additional annotations are imported if specified in the header. The collected data is then further processed in unique parameter types and identifiers for every type to create the functions for the command creation. The last step is the processing of the annotations. Every annotation gets replaced with its specified replacement. After everything is replaced the template for the output plugin is written.

The following description shows this process in more detail:

The generation starts with the reading of the header file by creating a list from the text lines. This list is given to `process_header.py` to extract the data from the text. Every Line is checked for commented sections, imports of additional configurations and for functions headers. If a `"/*"` is found any functions and imports are skipped over until a `"*/"` is found. A double slash before the function header means that the line is skipped because it is commented out. When an import is found the specified file is loaded and the lists `define_import`, `initialize_import`, `before_import`, `function_import` and `after_import` are added to the corresponding lists of the standard configuration. If a function is found it is first split into the function part and the following comment. The function is then further split into the return type, function name and the parameters which get split into parameter type and parameter name. All the information about the function gets saved into a class which is added to a list of functions which is returned by the `process_headers`. This function list is then given to `find_unique_types` to create a list of all used parameter types and returned this list. From this list the characters that specify the datatype are generated by `get_type_chars`. With the characters then the entries of the hashtable are generated that specify the types in the struct generation.

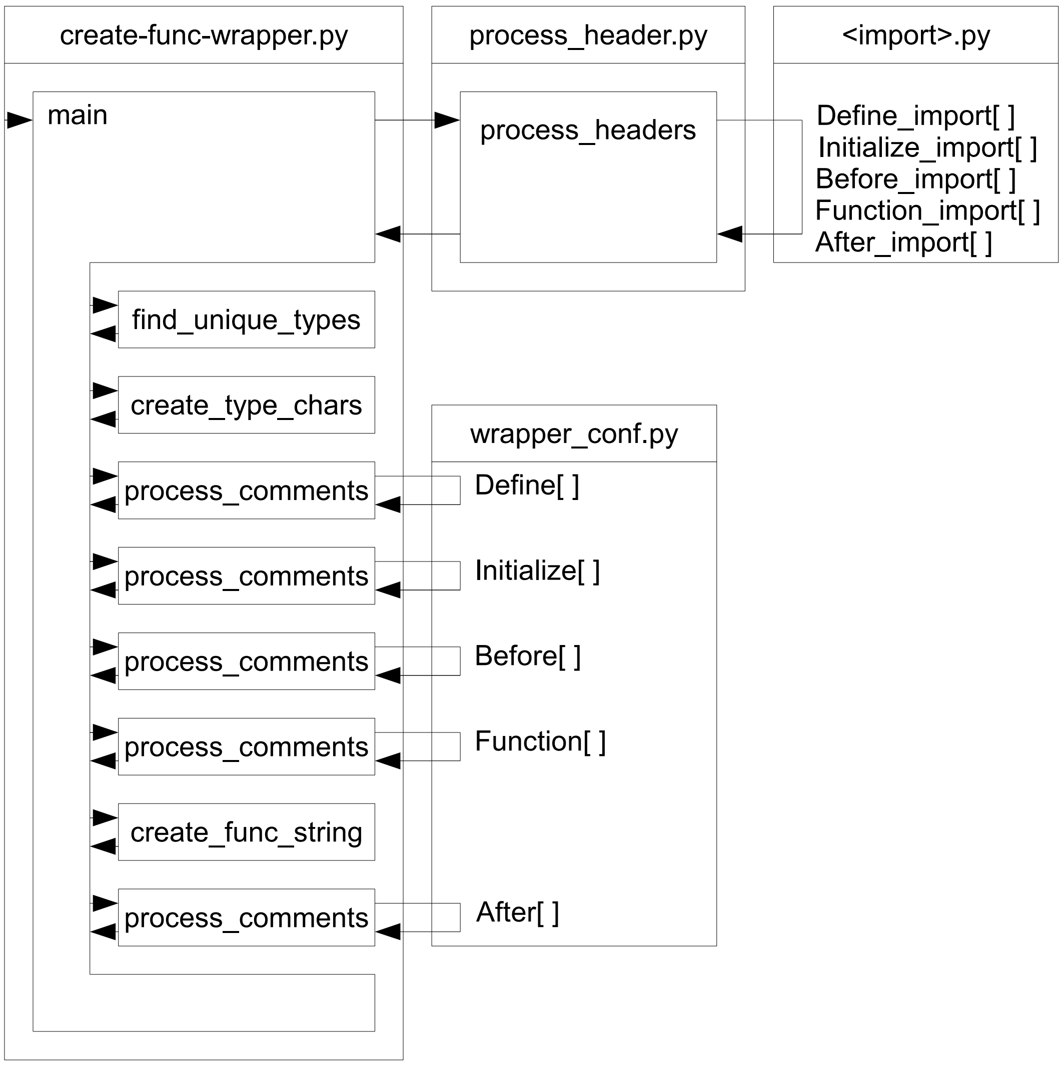

Figure 2.7: Diagram of the code generation through the wrapper

In the next step the writing function for the struct is generated from the characters and the types. Every character gets translated to a identifier in the switch to write a type of data into the struct. The operation that creates the string for error reporting is also created this way.

In the next step the comments are processed. Every comment gets replaced by the replacement specified in the configuration file. In this replacement are still placeholders for parameters and struct paths. This placeholders now get replaced by the parameters in the comment and the path of the struct and are then written to the output template file. This replacement is done for all of the five lists.

In this chapter we have seen the roles of the three plugin types: input, filter and output. They are responsible for the creation of data, filtering of the data and the processing, respectively. Data between plugins is transferred as C structures that are called commands. Commands contain the information belonging to a function including the needed interface. The plugins are managed by Tracereplay which handles the data flow. This is done with a thread for every input plugin and one thread for the output plugins.

To reduce work on the plugin creation templates have been created for the general structure. Additionally a generator script has been created to generate code from the function header. The generation includes helper functions for the command handling. To realize advanced code creation, annotations have been added to specify information for the creation process. In the next chapter the execution and output of Tracereplay will be shown.

3 Usage

This chapter shows how Tracereplay is executed and describes the available parameters. These are used to configure Tracereplay and their effect is explained. Furthermore, the generated statistic by Tracereplay is described.

3.1 Execution

This section shows how to execute Tracereplay and how to pass parameters to the program and the plugins. The syntax to execute Tracereplay is:

./Tracereplay <program options> <plugin option <options>>

It is important that the program options are given before the plugin options because every option after a plugin option is interpreted as a option for the plugin. The following options configure the behavior of Tracereplay:

- -b=X Sets the buffer size. Determines how many commands from the input plugins are buffered.
- -h Shows the help text with the available commands.
- –direct Deactivates the buffering of commands and the synchronisation to reduce the overhead. Only usable when one input plugin is used as synchronisation is not active.
- –strict Sets the strict error handling. When a warning occurs it is handled as an error and terminates the program after a warning.

The following options are for loading of the plugins. To run at least one input plugin must be loaded. Plugins that can act as input and output plugins have to be loaded with both parameters. Otherwise only one part is loaded. If more than one plugin of a type is loaded the order determines the order in which plugins are executed. Options for the plugin are given after the plugin name. The options i,f,o are reserved for the loading of plugins and cant be passed to a plugin.

- -i <name> <options> Loads the input plugin with the specified name.
- -f <name> <options> Loads the filter plugin with the specified name.
- -o <name> <options> Loads the output plugin with the specified name.

Example: `./Tracereplay -b=10 -i input_posix -r input.txt -o output_posix`
Sets the buffer size to 10 commands, loads the input plugin `input_posix` and passes the parameters ”-r input.txt” to the plugin and loads the output plugin `output_posix`.

3.2 Statistics

After a run is finished the statistic is printed with information about the commands that occurred during the run.

Listing 3.1 shows an example statistic which is printed at the end of a run. There are three topics in the statistic. The first shows the commands that were filtered out. Here a filter filtered out all commands belonging to MPI. The second shows commands that are executed by the loaded output plugins. The number shows the number of commands that were executed. If a command is executed by more than one output plugin it still counts as one command. The last topic shows failed commands. A command counts as failed if an output plugin supports the interface of the command but is missing the function for the execution or when an error was reported by the output plugin.

```
Filtered out commands:
  MPI_File_open: 1 times
  MPI_File_write: 2 times
  MPI_File_close: 1 times

Executed commands:
  open: 1 times
  read: 2 times
  write: 1 times
  close: 1 times

Failed commands:
  None
```

Listing 3.1: Example statistic of a finished run

This chapter has shown the execution of Tracereplay. It is possible to configure Tracereplay with parameters. They have to be specified before the parameters for the plugin loading. After a parameter for plugin loading is given the following parameters that are not for plugin loading are passed to the plugin. After Tracereplay is finished a statistic is shown. It gives information about commands that got filtered out, commands which were executed and commands that failed due to errors.

4 Creation of Plugins

This chapter shows how new plugins for Tracereplay are created. For every plugin type there is a template to start working with. In addition the usage of the wrapper and its annotation system is described to simplify the creation process.

4.1 Input Plugin

Listing 4.1 shows the source of the template for an input plugin. In the template the `initialize` function contains the allocation of the `local_vars` struct. Plugin variables come into the `local_vars` struct for thread safety. This struct is passed to `get_next_command` function on each call. The initialization is also the place where variables get initialized and where the options for the plugin are received. The options are given in a linked list to the plugin. The central function that needs implementation is the `get_next_command`. This function is called by the controller whenever a new command must be loaded into the command buffer. Its task is to generate the next command struct including required parameters. With the return of the created struct the function ends. If there are no more structs to be returned, this function has to return NULL to signal that the plugin is finished.

```
#include <stdio.h>
#include <stdlib.h>
#include <glib.h>
#include <gmodule.h>

struct local_vars {
// Place local variables here
};

typedef struct local * LOCAL_VARS_P;

G_MODULE_EXPORT extern LOCAL_P initialize(GSList * options) {
  LOCAL_VARS_P global = calloc(1,sizeof(struct local_vars));

  return global;
}

G_MODULE_EXPORT extern struct Command * get_next_command(LOCAL_VARS_P global) {
  //TODO: Command generation
}
```

Listing 4.1: Template for an input plugin

4.2 Filter Plugin

Listing 4.2 shows the template for a filter plugin. First part that needs to be changed is the interface to the interfaces that the filter should support. The `initialize` function is the place where the options for the plugin are passed to. The options are in a Linked List to be parsed by the plugin. For Tracereplay to get the supported interfaces the functions `get_num_interfaces` which returns the number of available interfaces and `get_interface` to receive an interface have to exist. The central function of a filter plugin is the filter function. When the filter is loaded, every struct gets passed to this function when the interface of the plugin equals the interface of the command. The filtering is then done by checking the parameters of the struct. The function returns a pointer to a command. This can be the command that was passed into the filter then no filtering happened. Other possibilities are to change parameters of the command and return the modified command, create a new command to replace the old one or to return NULL which filters the command out.

```
#include <stdio.h>
#include <stdlib.h>
#include <gmodule.h>

int NUM_INTERFACES = 1;
char * INTERFACE[] = {"*"}; //TODO: Specify Interfaces

G_MODULE_EXPORT extern void initialize(GSList * options) {

}

G_MODULE_EXPORT extern int get_num_interfaces() {
  return NUM_INTERFACES;
}

G_MODULE_EXPORT extern char * get_interface(int pos) {
  return INTERFACE[pos];
}

G_MODULE_EXPORT extern struct Command * filter(struct Command* s_command) {
  //TODO: Filter logic
}
```

Listing 4.2: Template for an filter plugin

4.3 Output Plugin

Listing 4.3 shows the beginning of the output template and an example for an output function. The interfaces specify which commands are supported by this plugin. This has to fit to the functions that can be executed. The `initialize` function is the place where the options for the plugin are passed to. The options are in a Linked List to be parsed by the plugin. For Tracereplay to get the supported interfaces the functions `get_num_output_interfaces` which returns the number of available interfaces and `get_output_interface` to receive an interface have to exist. Another function is `translate_command` which processes a command in case of an returned error by one of the command execution functions. Its task is to return a string with information about the parameters. If the generation was used the helper class contains the logic that fulfills this task. The most part of the output plugin are the execution functions which correspond to the command name. These receive the command that fits their name. Like the shown `out_example` which would take commands for the function example. This is where the logic for the execution is placed and the error checking is done. The last step is to add the needed imports for the functions that the plugin executes.

```
#include <stdio.h>
#include <stdlib.h>
#include <gmodule.h>
#include "commands.h"

int NUM_INTERFACES = 1;
char * INTERFACE[] = {"*"}; //TODO: Specify Interfaces
```

```
G_MODULE_EXPORT extern void initialize(GSList * options) {

}

G_MODULE_EXPORT extern int get_num_output_interfaces() {
  return NUM_INTERFACES;
}

G_MODULE_EXPORT extern char * get_output_interface(int pos) {
  return INTERFACE[pos];
}

G_MODULE_EXPORT extern char* translate_command(struct Command* s_command) {
  return get_command_string(s_command);
}

G_MODULE_EXPORT extern int out_example(struct Command* s_command) {
  int error = 0;
  example();
  return error;
}
```

Listing 4.3: Part of the template for an output plugin

4.4 Generation of Plugin Parts

To generate the complete template for the output plugin with all execution functions and the helper functions for the command handling, a header file of the functions that the plugin should support is needed. For the wrapper to work with the header file, it has to have a specific structure which is shown in Listing 4.4.

```
// POSIX
//import posix_conf.py
        int open(const char *pathname, int flags, mode_t mode);
        int close(int fd) //map fd fd;
```

Listing 4.4: Needed structure of a header file

The structure needs to fulfill the following requirements:

- Every function needs to be on a separate line
- Functions have to specify a return type
- Parameters need to have a type and name
- A `//import` comment and comments after functions have a special meaning as they are used by the annotation system.

4.4.1 The Annotation System

Listing 4.5 shows the output of the wrapper without the use of annotations. The generated output is only the function in the header file with its parameters and the return that all went well. To add additional code like the mapping of pointers, annotations are used in the header file. A double slashed comment after the function is added to achieve this. The first word is the name of the annotation followed by parameters for this annotation separated by spaces. It is also possible to add multiple annotations each beginning with double slashes. The annotations are parsed in the order in which they are specified.

```
G_MODULE_EXPORT extern int out_close(struct Command* s_command) {
  int error = 0;
  int func_return = close(s_command->u.s_close.fd);
  return error;
}
```

Listing 4.5: Generated code without the use of annotations

Listing 4.6 shows the defined annotations and the replacements. Additional annotations can be added here by adding elements to the list. To add an annotation it needs a name and separated by a comma the string that is generated by the wrapper. In this string the following placeholders are available:

- %par<num> inserts a parameter into the string. Parameters are given in the annotation after the annotation name. <num> is the number of the parameter beginning with one.
- %rline inserts all parameters in the comment that are not used by %par
- %struct inserts the path to the struct with the data of the function. Example: s_command->u.s_close

```
Define = ["define",  "GHashTable * %par1_table;",
               "createGlobalBuffer", "static char * globalBuffer = NULL;"]

Initialize = ["define","  %par1_table = g_hash_table_new(NULL, NULL);",
                   "createGlobalBuffer", "  globalBuffer = malloc(%par1);"]

Before = ["map", "  typeof(%struct.%par2) %par1 = (typeof(%struct.%par2))g_hash_table_lookup(%par1_table,
(gconstpointer) %struct.%par2);",
          "local_buffer", "  char * %par2_buff = malloc(10*1024*1024);",
   "timing", "  struct timespec *s_time;\n  struct timespec *e_time;\n clock_gettime(CLOCK_MONOTONIC, s_time);"]

Function = ["map",        "%par2 -> %par1",
            "local_buffer", "%par1 -> %par2_buff",
            "global_buffer", "%par1 -> globalBuffer"]

After = ["define", "  g_hash_table_insert(%par1_table, (gpointer) %struct.%par2, (gconstpointer) %par3);",
         "undefine", "  g_hash_table_remove(%par1_table, %struct.%par2);",
         "error", "  if(%par1) error = 1;",
         "null_error", "  if(%par1 == NULL) error = 1;",
         "local_buffer", "  free (%par2_buff);",
         "timing", "  clock_gettime(CLOCK_MONOTONIC, e_time);\n  printf(\"%f kb/s\\n\",func_return * 0.001 /
((e_time->tv_sec - s_time->tv_sec) + ((e_time->tv_nsec - s_time->tv_nsec)* 0.001 * 0.001 * 0.001)));"]
```

Listing 4.6: Annotations and their replacements defined in the wrapper_conf.py

Another annotation is the import of an additional configuration file. This is an annotation that comes at the beginning of a line. Its syntax is: //import <path> where <path> is the path of the configuration file. There are five lists for annotation definitions. Every list defines annotations which have influence on a different place of the generated source code. It is possible to add an annotation to more than one list to add code in multiple places. The following is a description of the five lists:

- Define: Inserts definitions to the beginning of the source file
- Initialize: creates code into the initialize function to do the plugin initialization
- Before: adds code before the function call defined in the header file
- Function: does replacements of parameters in the function
- After: adds code after the function call

In imported config files these list names end with `_import`. The generation is started by running `create_func_wra`
with the header file as the parameter. The wrapper creates the helper function (`commands.h` and `commands.c`) and Template for the output plugin (`output_template.c`) by applying the annotations.

This chapters has shown the creation of plugins. For every type of plugin a template with the needed functions has been shown and the functionality of these functions. Every plugin has a characteristic type of function. For input plugins it is the `get_next_command` which delivers the commands for Tracereplay. The filter plugin with the `filter` function which allows to pass, modify or restrict commands. And for output plugins the functions for each command with the logic to process the command.

To reduce needed work for the creation the generation is used. Here the needed structure of the header file has been shown. For the additional functionality details on how to work with the annotation system are given. This includes details on how to add additional annotations, the meaning of the different lists and how to import additional configurations.

5 Evaluation

This chapter evaluates the created framework with 3 examples. The first example is the creation of a new output plugin with the automatic generation script and tested with an input plugin. The created output plugin is then reused in the second example to build a new functionality with a new input plugin and a second output plugin. In the third example the performance of Tracereplay is tested. They show that the goals of this thesis are met.

5.1 Automatic Generation

To evaluate the automatic generation, a new output plugin is created. This plugin shall be able to execute a subset of the I/O functions defined by the POSIX standard. To begin, the header file of the functions is needed. In this example the header in Listing 5.1 is used.

```
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
int close(int fd);
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
off_t lseek(int fd, off_t offset, int whence);

FILE * fopen(const char *path, const char *mode);
FILE * fdopen(int fd, const char *mode);
FILE * freopen(const char *path, const char *mode, FILE * stream);
int fclose(FILE *fp);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE * stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fseeko(FILE *stream, off_t offset, int whence);
int fseek(FILE *stream, long offset, int whence);
int fsetpos(FILE *stream, fpos_t *pos);
```

Listing 5.1: Header file of the POSIX functions

Generating with the plain header file would create a plugin that directly calls the functions with the delivered parameters. As POSIX works with file descriptors and file pointers which are different in each run, the pointers given by the input plugin have to be mapped to the actual pointers that were created this run. This is needed because the input plugin does not know the pointers that were created during execution in the output plugin.

Normally annotations are added after the function, for a better readability in this example they are added a line above the function. To realize the mapping the `//define`, `//map` and `//undefine` annotations are added. The define annotation saves the pointer in a table to later map the parameter from the input plugin to the actual pointer. Undefine removes the pointer from the table on functions that close the file. Another sort of annotations are the `//local_buffer` and `//global_buffer`. These create buffers for the read and write functions to work with. The `//timing` annotation is used to measure the time and calculate the speed of the `fread()` function. The last part is the error checking to report errors that occur during the run. This is realized by the `//null_error` annotation, which checks the return value to be not null, and `//error` that defines statements to be checked. The prepared header file in Listing 5.2 is now ready for the wrapper.

```
        //define fd return_val func_return //createGlobalBuffer 10*1024*1024 //error ''func_return == -1''
        int open(const char *pathname, int flags, mode_t mode);
        //define fd return_val func_return //error ''func_return == -1''
        int creat(const char *pathname, mode_t mode);
        //map fd fd //undefine fd fd //error ''func_return == -1''
        int close(int fd);
        //map fd fd //local_buffer buf write //error ''func_return != (%struct.count)''
        ssize_t write(int fd, const void *buf, size_t count);
        //map fd fd //global_buffer buf //error ''func_return != (%struct.count)''
        ssize_t read(int fd, void *buf, size_t count);
        //map fd fd //global_buffer buf //error ''func_return != (%struct.count)''
        ssize_t pread(int fd, void *buf, size_t count, off_t offset);
        //map fd fd //local_buffer buf write //error ''func_return != (%struct.count)''
        ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
        //map fd fd //error ''func_return == -1''
        off_t lseek(int fd, off_t offset, int whence);

        //define fp return_val func_return //null_error func_return
        FILE * fopen(const char *path, const char *mode);
        //map fd fd //define fp return_val func_return //null_error func_return
        FILE * fdopen(int fd, const char *mode);
        //map fp stream //define fp return_val func_return //null_error func_return
        FILE * freopen(const char *path, const char *mode, FILE * stream);
        //map fp fp //undefine fp fp //error ''func_return != 0''
        int fclose(FILE *fp);
        //map fp stream //timing //global_buffer ptr //error ''func_return == (%struct.size)*(%struct.nmemb)''
        size_t fread(void *ptr, size_t size, size_t nmemb, FILE * stream);
        //map fp stream //local_buffer ptr write //error ''func_return == (%struct.size)*(%struct.nmemb)''
        size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
        //map fp stream //error ''func_return != 0''
        int fseeko(FILE *stream, off_t offset, int whence);
        //map fp stream //error ''func_return != 0''
        int fseek(FILE *stream, long offset, int whence);
        //map fp stream //error ''func_return != 0''
        int fsetpos(FILE *stream, fpos_t *pos);
```

Listing 5.2: Header file with annotations for the generation

To finish the plugin some last changes must be made to the automatically generated code in Listing 5.3. The commands.h is renamed to `commands_posix.h` to represent that the supported commands are only POSIX. The imports for the POSIX interface are added so that the functions can be executed and the interface of the plugin is set to POSIX.

```
#include <stdio.h>
#include <stdlib.h>
#include <glib.h>
#include <gmodule.h>
#include "commands.h"

//TODO: Place imports here

int NUM_INTERFACES = 1;
char * INTERFACE[] = {"*"}; //TODO: Specify Interfaces

GHashTable * fd_table;
GHashTable * fp_table;

G_MODULE_EXPORT extern void initialize(GSList * options) {
  fd_table = g_hash_table_new(NULL, NULL);
  fp_table = g_hash_table_new(NULL, NULL);
}

G_MODULE_EXPORT extern int get_num_output_interfaces() {
  return NUM_INTERFACES;
}

G_MODULE_EXPORT extern char * get_output_interface(int pos) {
  return INTERFACE[pos];
}

G_MODULE_EXPORT extern char* translate_command(struct Command* s_command) {
  return get_command_string(s_command);
}

G_MODULE_EXPORT extern int out_open(struct Command* s_command) {
  int error = 0;
  int func_return = open(s_command->u.s_open.pathname, s_command->u.s_open.flags, s_command->u.s_open.mode);
  g_hash_table_insert(fd_table, (gpointer) s_command->u.s_open.return_val, (gconstpointer) func_return);
  if(func_return == -1) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_creat(struct Command* s_command) {
  int error = 0;
  int func_return = creat(s_command->u.s_creat.pathname, s_command->u.s_creat.mode);
  g_hash_table_insert(fd_table, (gpointer) s_command->u.s_creat.return_val, (gconstpointer) func_return);
  if(func_return == -1) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_close(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_close.fd) fd = (typeof(s_command->u.s_close.fd))g_hash_table_lookup(fd_table,
(gconstpointer) s_command->u.s_close.fd);
  int func_return = close(fd);
  g_hash_table_remove(fd_table, s_command->u.s_close.fd);
  if(func_return == -1) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_write(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_write.fd) fd = (typeof(s_command->u.s_write.fd))g_hash_table_lookup(fd_table,
  (gconstpointer) s_command->u.s_write.fd);
  char * write_buff = malloc(10*1024*1024);
  ssize_t func_return = write(fd, write_buff, s_command->u.s_write.count);
  free (write_buff);
  if(func_return != (s_command->u.s_write.count)) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_read(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_read.fd) fd = (typeof(s_command->u.s_read.fd))g_hash_table_lookup(fd_table,
(gconstpointer) s_command->u.s_read.fd);
  ssize_t func_return = read(fd, globalBuffer, s_command->u.s_read.count);
  if(func_return != (s_command->u.s_read.count)) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_pread(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_pread.fd) fd = (typeof(s_command->u.s_pread.fd))g_hash_table_lookup(fd_table,
(gconstpointer) s_command->u.s_pread.fd);
  ssize_t func_return = pread(fd, globalBuffer, s_command->u.s_pread.count, s_command->u.s_pread.offset);
  if(func_return != (s_command->u.s_pread.count)) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_pwrite(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_pwrite.fd) fd = (typeof(s_command->u.s_pwrite.fd))g_hash_table_lookup(fd_table,
(gconstpointer) s_command->u.s_pwrite.fd);
  char * write_buff = malloc(10*1024*1024);
  ssize_t func_return = pwrite(fd, write_buff, s_command->u.s_pwrite.count, s_command->u.s_pwrite.offset);
  free (write_buff);
  if(func_return != (s_command->u.s_pwrite.count)) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_lseek(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_lseek.fd) fd = (typeof(s_command->u.s_lseek.fd))g_hash_table_lookup(fd_table,
(gconstpointer) s_command->u.s_lseek.fd);
  off_t func_return = lseek(fd, s_command->u.s_lseek.offset, s_command->u.s_lseek.whence);
  if(func_return == -1) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_fopen(struct Command* s_command) {
  int error = 0;
  FILE * func_return = fopen(s_command->u.s_fopen.path, s_command->u.s_fopen.mode);
  g_hash_table_insert(fp_table, (gpointer) s_command->u.s_fopen.return_val, (gconstpointer) func_return);
  if(func_return == NULL) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_fdopen(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_fdopen.fd) fd = (typeof(s_command->u.s_fdopen.fd))g_hash_table_lookup(fd_table,
(gconstpointer) s_command->u.s_fdopen.fd);
  FILE * func_return = fdopen(fd, s_command->u.s_fdopen.mode);
  g_hash_table_insert(fp_table, (gpointer) s_command->u.s_fdopen.return_val, (gconstpointer) func_return);
  if(func_return == NULL) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_freopen(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_freopen.stream) fp = (typeof(s_command->u.s_freopen.stream))g_hash_table_lookup(
fp_table, (gconstpointer) s_command->u.s_freopen.stream);
  FILE * func_return = freopen(s_command->u.s_freopen.path, s_command->u.s_freopen.mode, fp);
  g_hash_table_insert(fp_table, (gpointer) s_command->u.s_freopen.return_val, (gconstpointer) func_return);
  if(func_return == NULL) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_fclose(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_fclose.fp) fp = (typeof(s_command->u.s_fclose.fp))g_hash_table_lookup(fp_table,
(gconstpointer) s_command->u.s_fclose.fp);
  int func_return = fclose(fp);
  g_hash_table_remove(fp_table, s_command->u.s_fclose.fp);
  if(func_return != 0) error = 1;
  return error;
```

```
}

G_MODULE_EXPORT extern int out_fread(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_fread.stream) fp = (typeof(s_command->u.s_fread.stream))g_hash_table_lookup(fp_table,
(gconstpointer) s_command->u.s_fread.stream);
  struct timespec *s_time;
  struct timespec *e_time;
  clock_gettime(CLOCK_MONOTONIC, s_time);
  size_t func_return = fread(globalBuffer, s_command->u.s_fread.size, s_command->u.s_fread.nmemb, fp);
  clock_gettime(CLOCK_MONOTONIC, e_time);
  printf("%f kb/s\n\n",func_return * 0.001 / ((e_time->tv_sec - s_time->tv_sec) + ((e_time->tv_nsec -
s_time->tv_nsec)* 0.001 * 0.001 * 0.001)));
  if(func_return != (s_command->u.s_fread.size)*(s_command->u.s_fread.nmemb)) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_fwrite(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_fwrite.stream) fp = (typeof(s_command->u.s_fwrite.stream))g_hash_table_lookup(fp_table,
(gconstpointer) s_command->u.s_fwrite.stream);
  char * write_buff = malloc(10*1024*1024);
  size_t func_return = fwrite(write_buff, s_command->u.s_fwrite.size, s_command->u.s_fwrite.nmemb, fp);
  free (write_buff);
  if(func_return != (s_command->u.s_fwrite.size)*(s_command->u.s_fwrite.nmemb)) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_fseeko(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_fseeko.stream) fp = (typeof(s_command->u.s_fseeko.stream))g_hash_table_lookup(fp_table,
(gconstpointer) s_command->u.s_fseeko.stream);
  int func_return = fseeko(fp, s_command->u.s_fseeko.offset, s_command->u.s_fseeko.whence);
  if(func_return != 0) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_fseek(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_fseek.stream) fp = (typeof(s_command->u.s_fseek.stream))g_hash_table_lookup(fp_table,
(gconstpointer) s_command->u.s_fseek.stream);
  int func_return = fseek(fp, s_command->u.s_fseek.offset, s_command->u.s_fseek.whence);
  if(func_return != 0) error = 1;
  return error;
}

G_MODULE_EXPORT extern int out_fsetpos(struct Command* s_command) {
  int error = 0;
  typeof(s_command->u.s_fsetpos.stream) fp = (typeof(s_command->u.s_fsetpos.stream))g_hash_table_lookup(
fp_table, (gconstpointer) s_command->u.s_fsetpos.stream);
  int func_return = fsetpos(fp, s_command->u.s_fsetpos.pos);
  if(func_return != 0) error = 1;
  return error;
}
```

Listing 5.3: Generated output file from the wrapper

After the changes the beginning of the edited source code now looks like in Figure 5.4.

```
#include <stdio.h>
#include <stdlib.h>
#include <glib.h>
#include <gmodule.h>
#include "commands_posix.h"

//POSIX
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int NUM_INTERFACES = 1;
char * INTERFACE[] = {"POSIX"};
```

Listing 5.4: Last manual edits of the file

Now the source of the plugin is finished and the makefile can be adapted to the file names and the plugin can be maked. To test that the generated plugin works, an input plugin is needed. For this purpose a simple input plugin is created. The plugin should open a file, write to it and close the file. In Listing 5.5 an example is created which writes 45 bytes to a file.

```
#include <stdio.h>
#include <stdlib.h>
#include <glib.h>
#include <gmodule.h>
#include "commands_posix.h"

struct posix_local {
  struct Command * com_arr[5];
  int cur;
};

typedef struct posix_local * POSIX_LOCAL_P;

G_MODULE_EXPORT extern POSIX_LOCAL_P initialize(GSList * options) {
  create_hashtable("POSIX");
  POSIX_LOCAL_P local = calloc(1,sizeof(struct posix_local));
  local->cur = -1;
  //Prepare commands
  local->com_arr[0] = create_command("fopen","test.txt","w+",987);
  local->com_arr[1] = create_command("fwrite",0,1,40,987,40);
  local->com_arr[2] = create_command("fwrite",0,1,5,987,5);
  local->com_arr[3] = create_command("fclose",987,987);
  local->com_arr[4] = NULL;
  return local;
}

G_MODULE_EXPORT extern struct Command * get_next_command(POSIX_LOCAL_P local) {
  //return next command
  local->cur++;
  return local->com_arr[local->cur];
}
```

Listing 5.5: Code of the input plugin for testing

After making the input plugin, the framework is executed with the created input and output plugin. The execution shows no errors and the file test.txt was created during the run. A view on the file size shows a size of 45 bytes which is the result of the two writes with the lengths of 40 and 5 characters. This concludes that the plugins worked as intended.

5.2 Reuse of Existing Plugins

To evaluate the reusability of plugins a second input plugin is created that also uses the POSIX output plugin. This time the input plugin shall simulate the reading patterns on a file of an application. The syntax and functionality of the pattern is similar to IOZone. The input plugin in Listing 5.6 creates commands that do seek and read operations on the file that is specified with the -r parameter to Tracereplay. The commands depend on a pattern that is specified by -Y. The pattern contains 3 values which are the position in the file, the length to be read and the time to wait after the operation before the next operation begins. To realize the wait time a second output plugin shown in Listing 5.7 is created to take the sleep command. As there is only one thread for the output plugins the sleep stops the execution of commands. The input Thread is still active during this time which ensures a steady command supply.

```
#include <stdio.h>
#include <stdlib.h>
#include <glib.h>
#include <gmodule.h>

#include <unistd.h>
#include "commands_posix.h"

struct posix_local {
  char* file;
  FILE* pattern;
  char* buf;
  int action;
};

typedef struct posix_local * POSIX_LOCAL_P;

G_MODULE_EXPORT extern POSIX_LOCAL_P initialize(GSList * options) {
  create_hashtable("POSIX");
  POSIX_LOCAL_P local = calloc(1,sizeof(struct posix_local));
  local->buf = (char*)malloc(1024*sizeof(char));
  int i = 0;
  //Process plugin options
  while(g_slist_nth_data (options, i) != NULL) {
    char* option = (char*) g_slist_nth_data(options, i);
    //Set file that gets read
    if(strcmp(option,"-r") == 0) {
      i += 1;
      local->file = (char*) g_slist_nth_data(options, i);
    }
    //Set pattern file
    else if(strcmp(option,"-Y") == 0) {
      i += 1;
      local->pattern = fopen((char*) g_slist_nth_data(options, i), "r");
    }
    i += 1;
  }
  return local;
}

G_MODULE_EXPORT extern struct Command * get_next_command(POSIX_LOCAL_P local) {
  //When first call read pattern and open file
  if(local->action == 0) {
    local->action = 1;
    fread(local->buf, 1, 1024, local->pattern);
    return create_command("fopen",local->file,"r",1);
  }
  //When pattern is finished, this plugin is also finished
  else if(local->action == -1) {
    return NULL;
  }
  else {
    //Split first number from string
    char** split = g_strsplit_set(local->buf," \n",2);
    local->buf = split[1];
    int value = atoi(split[0]);
    switch(local->action) {
      //Create fseek command with specified offset
      case 1:
        local->action = 2;
        printf("offset: %d byte\n",value);
        return create_command("fseek",1,value,0,0);

      //Create fread command with specified length and as return value
      case 2:
        local->action = 3;
        printf("read: %d byte\n",value);
        return create_command("fread",NULL,1,value,1,value);

      //Create usleep command with specified time in ms and check for end
      case 3:
        printf("wait: %d ms\n",value);
        char** split = g_strsplit_set(local->buf," \n",2);
        local->buf = split[1];
        Check if pattern is finished
        if(split[0] == NULL) {
          break;
        }
        //Creates struct for usleep
        struct Command* newCommand = calloc(1,sizeof(struct Command));
        newCommand->type = "usleep";
        newCommand->interface = "THREAD";
        newCommand->u.s_usleep.time = value;
        return newCommand;

      default:
        return NULL;
    }
    local->action = -1;
    return create_command("fclose",1);
  }
  return NULL;
}
```

Listing 5.6: Code of plugin for pattern based IO operations

```
#include <stdio.h>
#include <stdlib.h>
#include <gmodule.h>
#include "commands_thread.h"

int NUM_INTERFACES = 1;
char * INTERFACE[] = {"THREAD"};

G_MODULE_EXPORT extern void initialize(GSList * options) {
}

G_MODULE_EXPORT extern int get_num_output_interfaces() {
  return NUM_INTERFACES;
}
```

```
G_MODULE_EXPORT extern char * get_output_interface(int pos) {
  return INTERFACE[pos];
}

G_MODULE_EXPORT extern char* translate_command(struct Command* s_command) {
  return get_command_string(s_command);
}

G_MODULE_EXPORT extern int out_usleep(struct Command* s_command) {
  int error = 0;
  usleep(s_command->u.s_usleep.time);
  return error;
}
```

Listing 5.7: Code of the sleep time plugin

The pattern in Listing 5.8 does two file operations the first reads at an offset of 100 bytes data of 100 bytes and waits for 1000 milliseconds. The second read starts at an offset of 1000 bytes and reads 200 bytes with a wait time of 1000 milliseconds after the read. This is also shown in the output visible in Listing 5.9. During the runtime the wait time of a second is visible because the print to the console is delayed. The output also shows the measured speeds of the read operations and the statistic shows that 2 fseek and 2 fread commands are executed which is what the pattern specified.

```
100 100 1000
1000 200 1000
```

Listing 5.8: Pattern used with the plugin in Listing 5.6

```
offset: 100 byte
read: 100 byte
wait: 1000 ms
17895.490336 kb/s

offset: 1000 byte
read: 200 byte
wait: 1000 ms
28632.784538 kb/s

Filtered out commands:
  None

Executed commands:
  fopen: 1 times
  fclose: 1 times
  fread: 2 times
  fseek: 2 times

Failed commands:
  None
```

Listing 5.9: Output of the pattern based input plugin

5.3 Performance

The replay framework should allow a rapid and low-overhead execution of commands, otherwise a recorded (or programmed) pattern is deferred by the framework and unable to reproduce intended behavior. Therefore, the performance of the framework is evaluated. As a reference a program with the structure of a plugin is created but instead of being called by the framework a simple loop gets the next command and calls the output function. Every 1000 executed commands the execution time is taken. Its taken every 1000 commands to reduce the impact of the time measuring on the results and to get an average value.

The observed timings are assessed in diagrams that plot timing per measurement; every square represents a measurement with the taken time per command. Average time is calculated by dividing the resulting time by 1000 (amount of commands per time measurement).

5.3.1 Measuring with Direct Calling

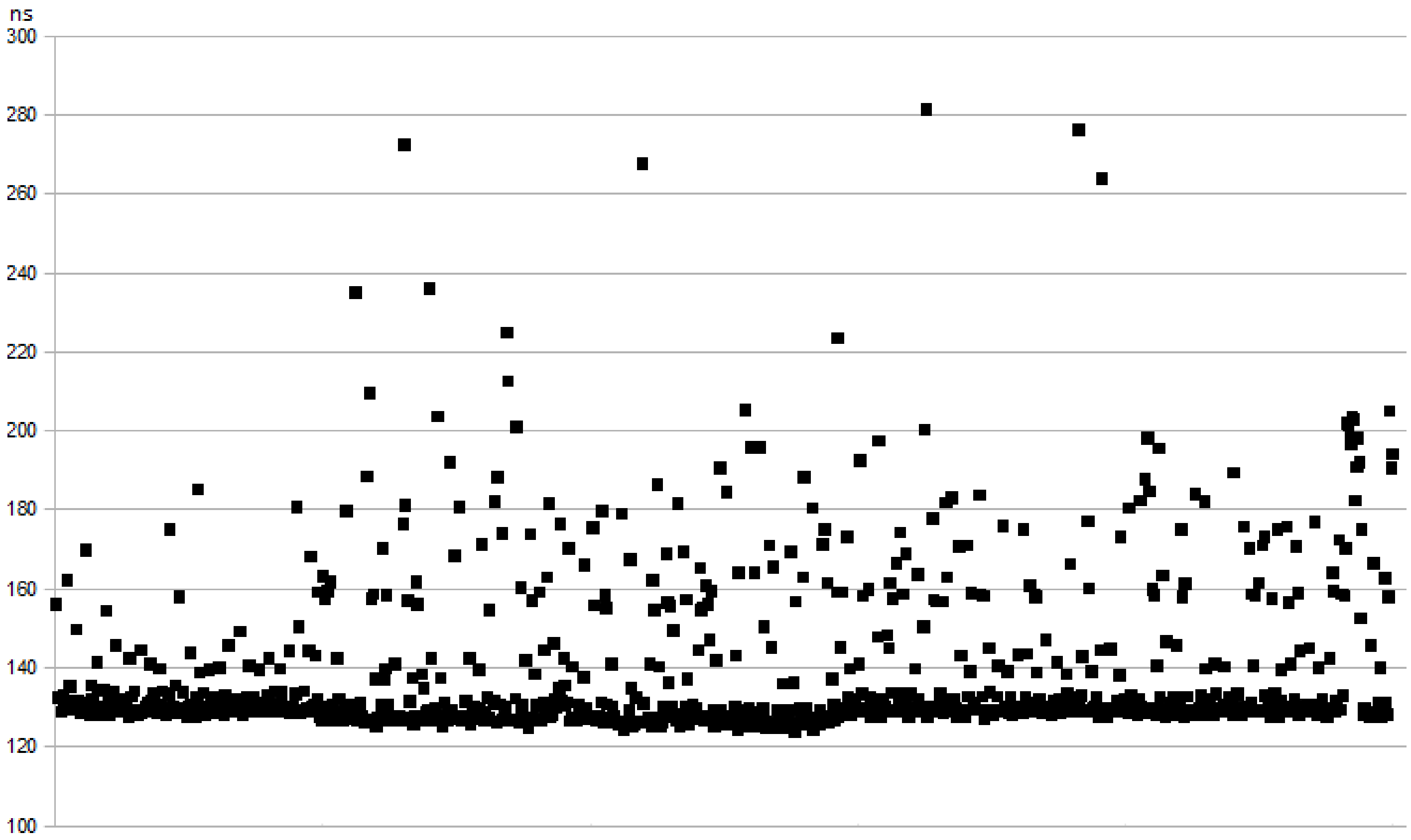

Figure 5.1: Execution times for one command measured by the program in Listing 5.10

Figure 5.1 shows the running times for the program in Listing 5.10 in nanoseconds. 1000 times were taken where every point shows the average execution time for 1 command. Many executions finished in 120 to 140 ns and most finished in up to 200 ns. Some took longer which could be the result of activity by the operating system. The average of all 1000 points is 140 ns.

```
#include "commands_timing.h"
#include <time.h>

int count = 0;
struct timespec *s_time;
```

```
G_MODULE_EXPORT extern void * initialize(GSList * options) {
  create_hashtable("TIMER");
  s_time = (struct timespec*)malloc(sizeof(struct timespec));
  clock_gettime(CLOCK_MONOTONIC, s_time);
  return NULL;
}

G_MODULE_EXPORT extern struct Command * get_next_command(void* local local) {
  return create_command("timing");
}

G_MODULE_EXPORT extern int out_timing(struct Command* s_command) {
  int error = 0;
  count++;
  //Measure time every 1000 commands
  if(count == 1000) {
    long time_nsec = 0;
    long time = 0;
    count = 0;
    time_nsec = s_time->tv_nsec;
    time = s_time->tv_sec;
    //Get time and print it
    clock_gettime(CLOCK_MONOTONIC, s_time);
    printf("%ld\n",((s_time->tv_sec - time) * 1000 * 1000 * 1000) + (s_time->tv_nsec - time_nsec));
    clock_gettime(CLOCK_MONOTONIC, s_time);
  }
  return error;
}

int main(){
        void * opt = initialize(NULL);
        int i=0;

  //Get commands and execute them
        for(i=0; i < 1000000; i++){
                struct Command * c = get_next_command(opt);
          out_timing(c);
        }
        return 0;
}
```

Listing 5.10: Speedtest for Direct Calling the Output Function

5.3.2 Measuring with Tracereplay in Direct Mode

To compare these values with the framework a plugin for time measurement is created. Listing 5.11 shows the plugin for measuring the execution times. It is nearly the same as the program in Listing 5.10. The difference now is that the control is handled by Tracereplay. So an additional counter was added to let the plugin finish once enough values are generated.

```
#include <stdio.h>
#include <stdlib.h>
#include <glib.h>
#include <gmodule.h>
#include "commands_timing.h"

#include <time.h>

int NUM_INTERFACES = 1;
char * INTERFACE[] = {"TIMER"};
```

```
struct posix_local {
};

int count = 0;
int count2 = 0;

struct timespec *s_time;

typedef struct posix_local * POSIX_LOCAL_P;

G_MODULE_EXPORT extern void initialize(GSList * options) {
  create_hashtable("TIMER");
  s_time = (struct timespec*)malloc(sizeof(struct timespec));
  clock_gettime(CLOCK_MONOTONIC, s_time);
  return NULL;
}

G_MODULE_EXPORT extern int get_num_output_interfaces() {
  return NUM_INTERFACES;
}

G_MODULE_EXPORT extern char * get_output_interface(int pos) {
  return INTERFACE[pos];
}

G_MODULE_EXPORT extern char* translate_command(struct Command* s_command) {
  return get_command_string(s_command);
}

G_MODULE_EXPORT extern struct Command * get_next_command(POSIX_LOCAL_P local) {
  //Finish after 1000 measured times
  if(count2 == 1000) {
      return NULL;
  }
  else {
    return create_command("timing");
  }
}

G_MODULE_EXPORT extern int out_timing(struct Command* s_command) {
  int error = 0;
  count++;
  //Measure every 1000 commands
  if(count == 1000) {
    count2++;
    long time_nsec = 0;
    long time = 0;
    count = 0;
    time_nsec = s_time->tv_nsec;
    time = s_time->tv_sec;
    //Get time and print it
    clock_gettime(CLOCK_MONOTONIC, s_time);
    printf("%ld\n",((s_time->tv_sec - time) * 1000 * 1000 * 1000) + (s_time->tv_nsec - time_nsec));
    clock_gettime(CLOCK_MONOTONIC, s_time);
  }

  return error;
}
```

Listing 5.11: Speedtest through the framework

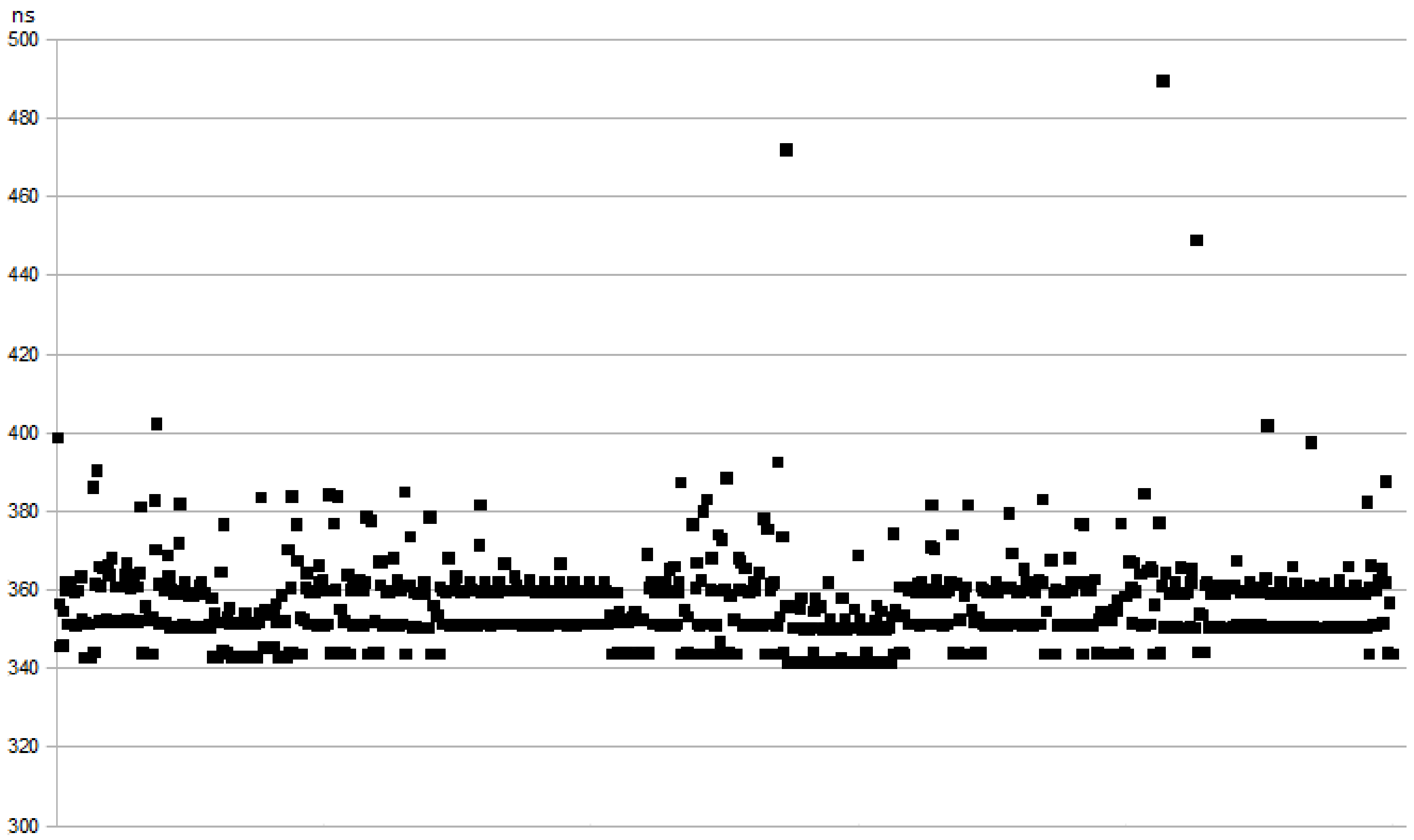

Figure 5.2: Execution times for one command measured by the plugin in Listing 5.11 with direct flag

Figure 5.2 shows the running times measured with Tracereplay in direct mode. In this mode no buffering of commands is done and no synchronisation between plugins. Compared to Figure 5.2 the execution times are longer because of the additional functions like filtering and error checking. Most times are between 340 and 370 ns with some results a bit over 400ns. The average lies at 354 ns.

5.3.3 Measuring with Tracereplay in Standard Mode

Figure 5.3 shows the running times measured with the framework in standard mode. In this mode the buffering is active and writes to the buffer have to be synchronized. Compared to Figure 5.2 the execution times are much longer which is a result of the synchronisation. With most times around 2800 ns to 3100 ns. The average is at 2877 ns. Even when the times are higher they were measured with a very simple command generation that needs very short times to execute to measure the overhead. In the real usage the generation will be more complex and more time intensive which is the case in which a parallelization is needed. Then the parallelization of multiple input plugins will make up for a part of the added overhead through the synchronisation.

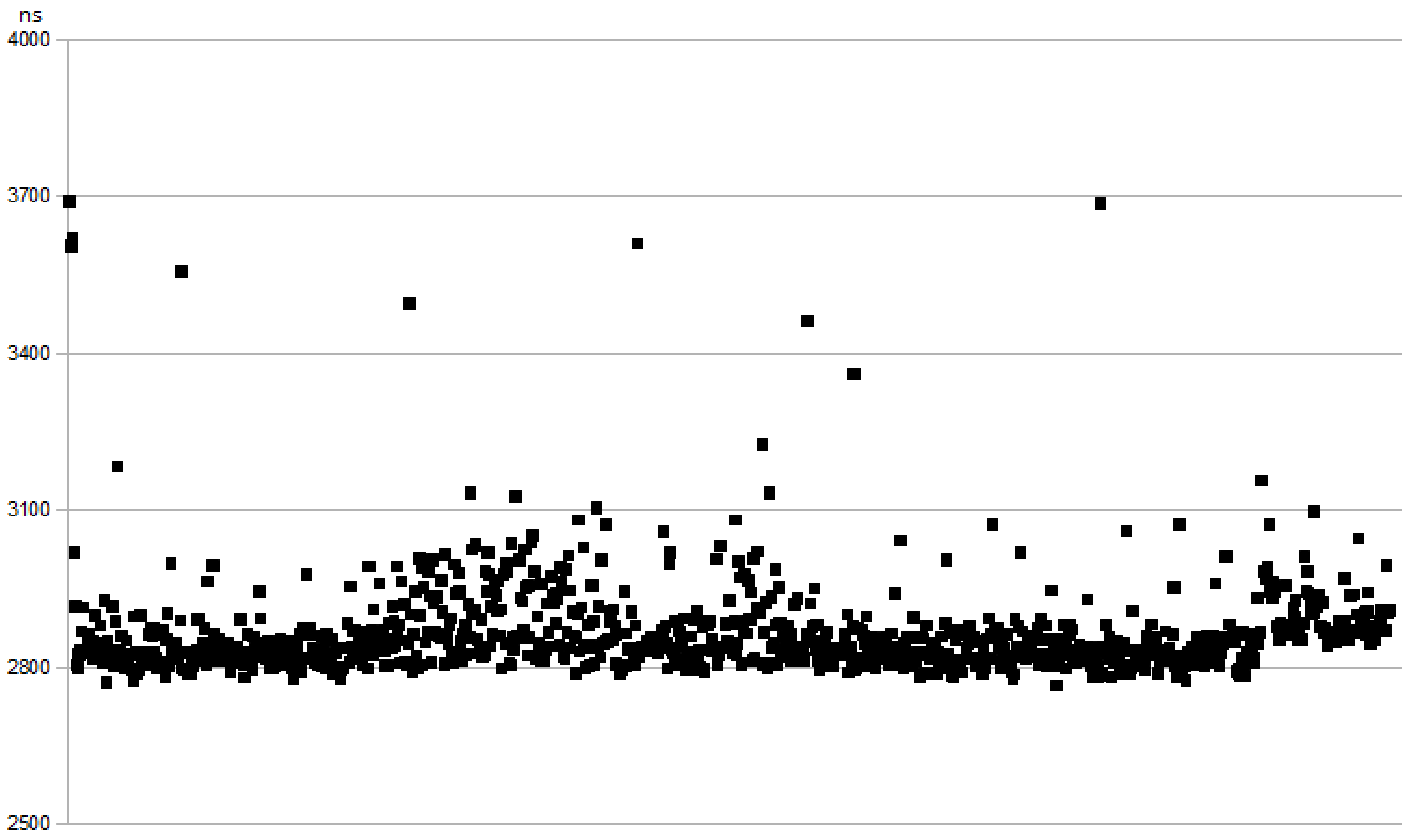

Figure 5.3: Execution times for one command measured by the plugin in Listing 5.11 without flags

The evaluation has been done with examples for the POSIX standard. An output plugin has been created with the generation script together with manual editing. This plugin has been tested with a simple input plugin to show that the generation process works. In the next step this output plugin has been used with another input plugin and an additional output plugin to create a pattern based benchmark. This has shown the reusability of plugins. The last Test was the performance test. Here the performance of Tracereplay was compared with a program that directly calls the plugin functions. The result has shown that an overhead exists and that the parallelization makes up for a part of it.

6 Summary and Future Work

6.1 Summary

The idea of this thesis is to reuse recurring parts in the workload processing. For this purpose a plugin based framework has been designed.

Plugins are classified into three classes: input, filter and output. These are responsible for the creation of data, filtering of the data and the processing, respectively. Thereby data is organized in structures containing data for one function. This includes information about the interface the function requires.

The framework handles plugins and controls the data flow. It allows loading of multiple plugins which then work together with plugins that support the same interface. This enables us to use created plugins for different tasks by using them together with other plugins to create a new functionality.

As the general structure of some plugins is always similar, templates have been created that reduce the work in creating new plugins. Another approach to make the creation easier is to automate parts of the creation which depend on the header of the used functions. For this purpose a code generator has been designed which generates plugin functions and functions for data handling based on the function structure. To enable advanced code generation an annotation system has been created. This system allows the addition of semantical information to generate functionalities which can not be created from the function structure alone.

Finally this work has been evaluated with examples for the POSIX standard. A plugin was created with the code generator and used together with different plugins to create different functionalities. These have shown that the generation is working and that plugins are reusable. Moreover the performance of the program was analyzed. The result is that an overhead exists but that it gets partly compensated through the added parallelization.

6.2 Future Work

As a piece of software is never completely finished there is always room for improvements. This section shows ideas to further improve the Tracereplay software framework.

At the moment annotations for the generation have to be written after the function in the header file. A way to improve this would be to make it possible to write them above the function like how it is shown in Listing 5.2. This would improve the readability of the prepared header file.

Another idea is to allow the creation of additional placeholders like the existing %struct. These are hard-coded in the current version. This would give more flexibility in the generation and allows to shorten often used phrases with a placeholder.

To improve speed when more than one input plugin is used a double buffer per input can be created. This has been started but did not finish during the time of this thesis. Every input plugin then uses two buffers and writes to their own buffer. When the buffer is full and if the other buffer is empty, both buffers are switched. The full buffer is then read by the thread for the output plugins or added to a waiting list when another buffer is already read. With double buffering the synchronisation to write every command into the buffer is no longer needed. Instead synchronisation is used when switching buffers. This reduces the overhead arbitrarily as the buffer size can be adjusted. However, it adds waiting time during the start phase until the first buffer is full.

To support more programming interfaces additional plugins are needed. This is the largest work for the future: to create more plugins.

A Appendix

A.1 Error Messages

- no input plugin used. you need to use at least one input plugin with: -i <pluginname>:
  Indicates that there was no input plugin used. The framework can not run without a source of workloads.
- could not find/access input and output plugins in "plugins/input_output":
  Either the folder plugins/input_output/ was missing and got created or there are no access rights to the folder. In that case the access right have to be given before the execution is possible.
- input plugin <name> not found:
  filter plugin <name> not found:
  output plugin <name> not found:
  The plugin could not be found. It is either misspelled or not in the right folder.
- plugin <name> does not support additional options:
  The plugin with the given name does not support options because it is missing the set options function.
- Could not find function initialize in <name>:
  The plugin with the given name is missing the initialize function. The execution of the initialization is skipped.
- Input Plugin <name> is missing get_next_command(). Plugin skipped!:
  The input plugin misses the function for the reading of the commands. As this is a critical function for an input plugin the plugin is skipped.
- Unable to create Thread. Error Code: <number>:
  An error occurred in the creation of the thread and the error number returned by pthread_create is given. This error is fatal and aborts the run.
- <name> does not offer a valid filter function:
  The given filter plugin is missing the filter function and can not be used. The plugin is skipped.
- <name> returned error on <function> with error code <number>. Command was <command>, interface: <interface> <parameters>:
  The given output plugin reported an error during runtime. The command name, interface and the used parameters are shown. For details on the error number check the plugin documentation.
- <name> does not support <command>:
  The command given to the output plugin is not supported. This indicates that the plugin got a wrong command, does not support all commands of the supported interface or uses the wrong interface.
- Direct command processing is only available with one input plugin:
  direct processing mode was used with more than one input plugin. This is not possible because synchronisation between the plugins is needed which is not given in direct mode.

- internal error: <error number>
  Indicates an error in the argument processing. Should not occur under normal circumstances as it is an error in the programming.

Listings

List of Figures

Bibliography

[1] Benchmarking Computers and Computer Networks. Online: `http://www.ict-fire.eu/uploads/media/Whitepaperonbenchmarking_V2.pdf`.

[2] EZTrace: Easy to Use Trace Generator. Online: `http://eztrace.gforge.inria.fr/`, 2012.

[3] TOP500 Supercomputing Sites. Online: `http://www.top500.org/`, 2012.

[4] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[5] Jason Jacobs. Synthetic VS Real World Benchmarks. Online: `http://www.techwarelabs.com/articles/editorials/real-vs-synthetic/`, 2008.

[6] K. Jeffay. The Real-Time Producer/Consumer Paradigm: A Paradigm for the Construction of Efficient, Predictable Real-time Systems. In *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice*, pages 796–804. ACM, 1993.

[7] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang Nagel. Introducing the Open Trace Format (OTF). In Vassil Alexandrov, Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3992 of *Lecture Notes in Computer Science*, pages 526–533. Springer Berlin / Heidelberg, 2006.

[8] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools*, pages 139–155. Springer, 2008.

[9] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak. The PAPI cross-platform interface to hardware performance counters. In *Department of Defense Users' Group Conference Proceedings, Biloxi, Mississippi, June 18*, volume 21, page 2001, 2001.

[10] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard – Version 2.2. Technical report, September 2009.

[11] Hans Werner Meuer. The TOP500 Project: Looking Back over 15 Years of Supercomputing Experience, 2008.

[12] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In *Parallel Computing: Architectures, Algorithms and Applications, volume 15 of Advances in Parallel Computing*, pages 637–644. IOS Press, 2007.

[13] Olga Mordvinova, Dennis Runz, Julian Kunkel, and Thomas Ludwig. I/O Performance Evaluation with Parabench – Programmable I/O Benchmark. *Procedia Computer Science*, pages 2119–2128, 2010.

[14] G.J. Narlikar and G.E. Blelloch. Pthreads for dynamic and irregular parallelism. In *Supercomputing, 1998. SC98. IEEE/ACM Conference on*, pages 31–31. IEEE, 1998.

[15] R.H.B. Netzer and B.P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[16] W.D. Norcott and D. Capps. IOZone Filesystem Benchmark. Online: `http://www.iozone.org/`, 2012.