

IOProf

Thomas William
thomas.william@zih.tu-dresden.de

March 28, 2007

1 motivation - why ioprof

The goal is to provide the users with a tool to monitor the io-activity. An important premise was that the monitoring should be possible without altering the application, which simply may not be possible (no sources, only binary applications). The monitoring data should be file centric - meaning that the data is not gathered as input/output rates but for each accessed file separately. It was written in such a way that it is possible to write different logging-functions.

2 iotrack

Ioprof is based on a project named iotrack, started by Per Ekman and Philip J. Mucci, who both worked at that time at the Center for Parallel Computers (PDC) at the Royal Institute of Technology in Stockholm, Sweden. It never reached beyond beta status (in the authors view) and was released under LGPL (since it incorporates code from glibc) for this thesis. The sources are available at the cvs server of the Open Source Performance Analysis Tools project (OSPAT¹). The source code was stripped of all unnecessary parts. It was restructured to make different logging-backends possible. Therefore larger parts of the original functionality of iotrack could be preserved. Great effort was put into the documentation of ioprof (including the iotrack parts) which was missing nearly completely (only a README containing the compile & run instructions are shipped with iotrack).

This thesis uses the part of iotrack that does the library loading depending on the preload-mechanism (see 4.1) available on all x86-systems. It was also tried to preserve the interface the simple printf backend uses to do the logging. This resulted in only minor changes which had to be done to use the printf backend from iotrack under ioprof. Michael Kluge from ZIH made some add-ons including a different backend. This backend was also incorporated into ioprof.

¹<http://www.ospat.org>

3 sourcecode-structure

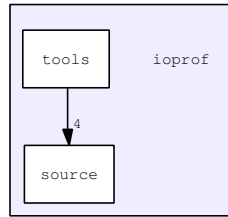


Figure 1: structure of the ioprof source-code directory

This section gives an overview over the project from a programmers point of view, illustrating the structure and call-hierarchy of the files. The functional analysis follows on page 5.

3.1 core

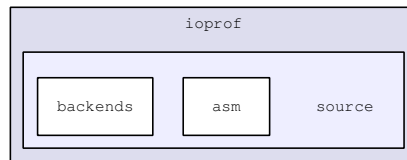


Figure 2: structure of the source directory

The core consists of the files handling the loading of the library which includes all the wrappers necessary to trap the I/O-functions in the libc-library and of course a possibility to do I/O without trapping itself. If there would be no circumvention to the logging mechanism the ioprof-library would try to trace itself resulting in an infinite recursive loop.

libload.h

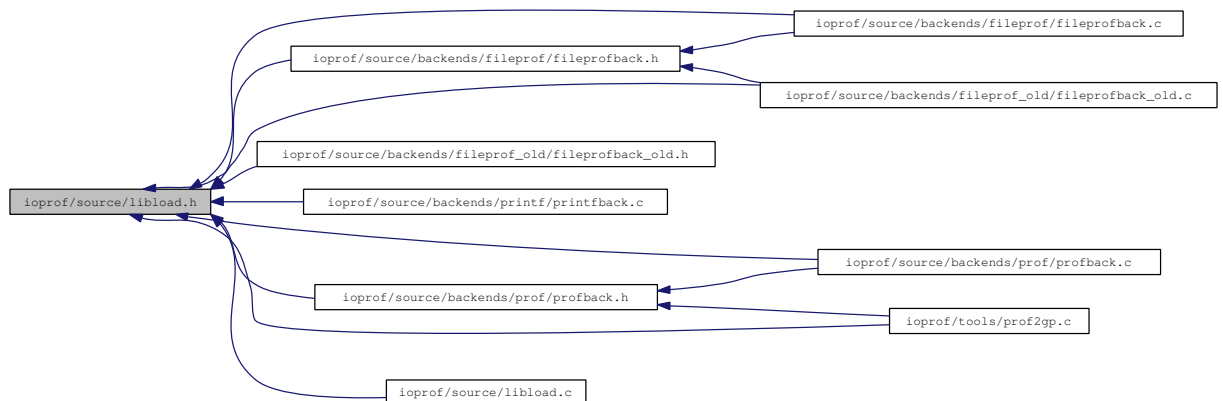


Figure 3: list of files depending on libload.h

The first element you will discover in libload.h is an enumerate type called `func_id` which is used to associate all the trapped functions with an integer. Now each function can use the corresponding integer instead of the name to refer to a special I/O-function, thus not only saving

time and memory, it also eases the writing of new backends because you can use large case-switches to act different on certain I/O-functions. To further easy this process a char-array function_types is defined and later filled in iowrap.c (see 3.1) which assigns a type to each of these I/O-functions (see 3.1 page 3):

- I/O_READ_FUNCTION
- I/O_WRITE_FUNCTION
- I/O_POS_FUNCTION
- I/O_ADMIN_FUNCTION
- OTHER_FUNCTION

There are 2 other arrays, fnames is a stringarray holding the real names and farray is a container for the functionpointer.

A larger part of the file libload.h is dedicated to defines calling the I/O-functions of the libc directly and thus preventing the logging: As you can see it uses the just mentioned farray to call the functionpointer of the open()-function defined in the libc. These REAL_* defines are used to do the real I/O in the before functions after the logging data is obtained.

libload.c

The libload.c contains the function libload() which is called upon library loading. This function tries to load the libc via dlopen() and then fills farray with the help of dlsym. That is how the functionpointer used in the REAL_* defines are gathered. Also some additional information regarding the application initiating the library are obtained, for example the executable name and its arguments. Libload.c also contains a function called emergency_perror() which is needed because at this point all normal I/O calls are trapped, but not yet loaded. This helper prints a string describing the source of the emergency in case of any problems at this early stage.

iowrap.c

The file iowrap.h only consist of defines, some of them handle special issues with the gnu compiler and the other defines have already been mentioned above. iowrap.c starts with two char arrays, function_types and fnames defining the type and names of the different log-able functions provided by libc. Despite some additional helper functions like a function for printing a backtrace in case of problems this file does exactly was the name states. It holds the wrapper functions for all trapped I/O calls.

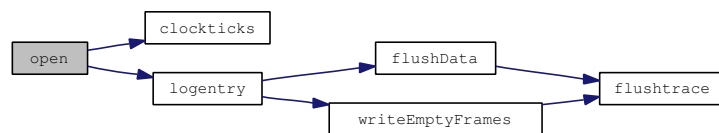


Figure 4: example of a captured call

These wrapper parse the arguments, use the REAL_* mechanism to call the I/O function and then call the logentry function provided by the backend. Here is an example of such a logentry() call for open():

```
1 logentry(OPEN_I, t0, clockticks(), fd, 0, 0, pathname);
```

Listing 2: define calling the I/O-function

As you can see, internally the integer representation of open is used. The logentry also contains the time the I/O call occurred (t0) and the time the call finished (clockticks()). Both, the real name (pathname) and the filedescriptor (fd) in form of an integer are needed to later associate the read() and write() calls only using the fd to a filename.

callmacros*.c

The two files, callmacros-ia64-linux.h and callmacros-x86-linux.h, are needed because certain functions are handled different on these platforms. On ia64 socket calls are socket calls, whereas on ia32 socket-related calls are all mapped to sys_socketcall which takes an argument specifying the actual call with all arguments stored in an array. In the current version of ioprof as a library these macros are not needed, they are solely there for future enhancements which re-enable a feature present in iotrack to use this tool as an application like strace.

backend.h

Here everything necessary to implement a backend is gathered. This includes a forward declaration of the logentry() function as well as the implementation of the function clockticks() which is used to acquire the timestamps. This function uses rdtsc() on ia32 and the proper registers on ia64 respectively to obtain the timestamps. This may change in the future as upcoming technics like foxtan may render this implementation unusable. But because the timekeeper is encased in this function, replacing the underling function is easy (see 5 for further details). The remaining part of the file is dedicated to useful defines for backend-programmers.

3.2 backends

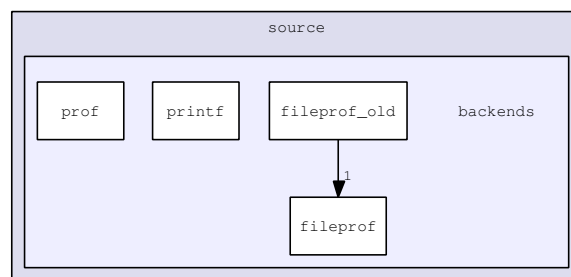


Figure 5: different backends contained in the backends-folder

A backend has to implement three functions, init_backend(), fini_backend() and the logentry() function.

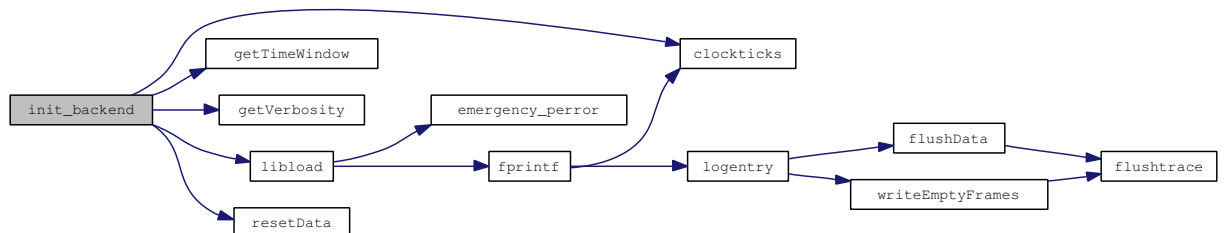


Figure 6: init is called upon initialisation of the application

Init_backend() is called at the beginning of the logging process and fini_backend() is called at the exit of the application respectively. Logentry() is called each time an I/O call occurs in the process the profiling library is running.

3.3 tools

The tools folder is a conglomeration of programs used by the different backends to parse the generated log-files. It also includes some applications needed by the lcg-mon-wn scripts as well as some test applications used during development.

4 working principle

The requirements of the monitoring facility were a transparent usage with already compiled programs. Therefore all approaches using sourcecode altering like recompiling with debugging information or special pragmas in the code were not feasible. The approach iotrack and ioprof use is the usage of the environment variable LD_PRELOAD.

4.1 The dynamic linker and the LD_PRELOAD mechanism

The dynamic linker is the part of an operating system that loads and links the shared libraries for an executable when it is run. The linker (typically) make use of a shared library itself which may alter the behaviour of the executable. This dynamic linker shared libraries vary between the systems - but in the context of LCG and Scientific Linux it can always be found under /lib/ld-linux.so.2. An additional impact on the behaviour can be achieved based on a common set of environment variables, including LD_LIBRARY_PATH and LD_PRELOAD. LD_PRELOAD instructs the linker to load additional libraries into a program, beyond what was specified when it was compiled. It allows users to add or replace functionality for good or worse. This is the reason for programs running setuid to ignore LD_PRELOAD (on most linux flavours).

The user can imagine a dynamically linked program as being incomplete. The missing parts are loaded at runtime by the dynamic linker. The tool ldd can be used to print shared library dependencies. A simple program written in C needs at least ld-linux.so and the standard C library libc.so.6

```
1 bash: ldd sob
2   libc.so.6 => /lib/tls/libc.so.6
3   /lib/ld-linux.so.2 => /lib/ld-linux.so.2
```

Listing 3: minimal dependencies

4.2 GNU C Library

The C language provides no built-in tools for performing common operations like I/O, memory management, string manipulation and the like. Instead, these facilities are defined in a library, which is compile and link with applications. This work is interested in the part of the libc dealing with I/O:

I/O on Streams which covers the high-level functions that operate on streams, including formatted input and output.

Low-Level I/O which covers the basic I/O and control functions on file descriptors.

File System Interface which covers functions for operating on directories and for manipulating file attributes such as access modes and ownership.

Pipes and FIFOs which includes information on the basic interprocess communication facilities.

Sockets which covers a more complicated interprocess communication facility with support for networking.

While the mechanisms used in ioprof are capable of capturing every functionality provided by the libc, the focus in the LCG context is on I/O functions and specifically on the low-level I/O:

Opening and Closing Files How to open and close file descriptors.

I/O Primitives Reading and writing data.

Stream-level I/O is more flexible and usually more convenient. Therefore, programmers generally use the descriptor-level functions only when necessary. So in order to extract the information necessary for monitoring the data in a file centric way, additionally the I/O on streams functionality of libc had to be traced:

Opening Streams How to create a stream to talk to a file.

Closing Streams Close a stream when you are finished.

Simple Input/Output Unformatted input and output by characters and lines.

Block Input/Output Input and output operations on blocks of data.

The actual implementation supports logging for the following functions:

open()	fork()	rewind()	puts()	recv()
open64()	pread()	fsetpos()	ungetc()	recvfrom()
read()	pwrite()	creat()	vfscanf()	recvmsg()
write()	fdopen()	creat64()	fscanf()	execve()
close()	fopen()	fgetc()	vfprintf()	execl()
lseek()	fopen64()	fgets()	fprintf()	execlp()
dup()	fclose()	getc()	socket()	execle()
dup2()	fread()	gets()	accept()	execv()
readv()	fwrite()	fputc()	send()	execvp()
writelv()	fseek()	fputs()	sendto()	
fcntl()	fseek64()	putc()	sendmsg()	

4.3 backends

Everything described so far can be seen as a framework on which to build your own profiler - called backends in this context. Iotrack comes with a really simple example. The only available backend in iotrack prints the called function to the log-file. Another implementation was written by Michael Kluge and uses a proprietary binary format to save its log-data. His version dumps all I/O activity at adjustable time intervals. It records for each time interval the total number of bytes read and written, the number of calls to each I/O function and the used block sizes for I/O (records for each write/read access to the power of 10 of the used block size). These log-files can be converted afterwards to a gnuplot readable file. The needed application is called prof2gp and is available in the tools folder. Michael Kluge also wrote a QT-based GUI for these log-files. The third backend was written within the scope of this thesis for the LCG monitoring.

The one major change to the interface (necessitating changes to the other backends) was the addition of one argument to the logentry() function adding the filename to be written to the log-file. This was done in order to later associate the I/O to the different files being accessed. A lot of work was put into automating the process of log-file generation because the library has to run in heterogeneous environments at LCG and thus has to be self error-correcting. The other backends expect the log-files to be examined after the measurement run. At LCG the log-files

have to be parsed while there is still data added. In long running jobs its nessessary to decide which log-files are still active and which are already closed. Otherwise each time the monitoring process needs I/O data, all log-files would have to be parsed which may take a very long time even exceeding the time between two polls from the monitoring process. To explain all the steps done by the fileprof backend the next paragraph describes them in chronological order for a better understanding.

When the library is loaded the `init_backend()` function is called. It first checks and possibly creates two folders, `ioprof` and a subfolder called `sl`. Then the name of the log-file is generated and the file is opened. Each (sub)process being monitored writes into its own log-file, therefor each log-file has its process-id as a suffix. The next step is to gather some general information composing the header of a log-file. Here is an example of how the result looks like:

```

1 Time:1172198764:905090:13952109135244756
2 Name:/opt/exp_soft/ghep/dist/releases/Moose-18.6.3i/bin/MooseApp
3 Args:/Moose.tcl 16966
4 TpS:21943238
5 PID:22209
6 PPID:22075

```

Listing 4: define calling the I/O-function

To relate all the timestamps to a real date, the first line contains the seconds and microseconds since the first january 1970. The last value of this triple is the corresponding timestamp. The next two lines contain the name of the application being logged as well as its arguments. `TpS` is an abbreviation for ticks per seconds and is used together with the `Time` of the first line to compute the real world time from the individual timestamps of each log-file entry. A process can inherit open filepointer from its parent process. If there is an I/O call which can not be assigned to a filename, the `PPID` can be used to determine the filename from the log-file of the parent process (although this is currently not implemented due to the overhead this would cause). As was mentioned before, there is a strong need of knowing which log-files are already finished and which are still written to. Another task is to gather the data from multiple log-files simultaneously to compute the results in the monitoring process. It would be very timeconsuming to open each file to get to know wether it is still actively written and when to queue it in the monitoring process. The solution to these issues was created with the help of symbolic links. For that purpose the subfolder `sl` (symbolic links) was created. For each new log-file an symbolic link to that file is created by the `init_backend()` function in the `sl` subfolder. The symbolic link starts with the clocktick used in the first line (`Time`) of that file. Therefore it is unnessessary to open the files in order to sort them. The timestamp is followed by an underscore as a separator from the next number - the clocktick marking the end of that log-file. A log-file still being written to has a 0 written as a endtime which makes it easy to spot closed and still open log-files. After the symlink has been written the `init_backend()` has finished its work. Now each intercepted I/O function triggers a call to `logentry()` which produces one additional line in the log-file:

```

1 open:14067122678612063:14067122678623045:0:0:7:FrameScripts/talkto.tcl
2 read:14067122678667856:14067122678671207:922:0:7
3 close:14067122678707079:14067122678708671:0:0:7

```

Listing 5: snippet of a log-file

This is rather unefficient and therefore some internal buffering is done to minimize the I/O caused by the profiling. In addition, these log-files are very inefficient concerning their size but are human readable at the present state - this may change to improve performance. When the process is closed because the application has finished, a timeline like the one from the beginning is written to the log-file - this make sanity checks possible to ensure the accuracy of the timestamps. When everything went right the last line in a log-file is `endoflog-file` stating that everything went right and the log-file was closed properly - this is done in the `finalize` function `fini_backend()`.

Now the symbolic link is removed by a new one containing the correct endtime and the logprocess is finished.

5 timemeasurements

The First Version of ioprof simply used `gettimeofday()` to measure the start and end time of an I/O function. The first measurements and benchmarks showed that depending on the underlying algorithm to be logged, a fairly big amount of the overhead generated by ioprof was introduced by `gettimeofday()`. So alternate ways of time-measurement were investigated. A first step was to move the functionality into an own function so that it was made easier to replace the underlying timer in the future.

5.1 The jiffies Counter

The linux kernel keeps track of the flow of time by means of timer interrupts. Timer interrupts are generated by the system's timing hardware at regular intervals according to the value of `HZ`, which is an architecture-dependent value defined in `linux/param.h`. There is a counter called `jiffies` which is initialized to 0 at system boot, so it represents the number of clock ticks since last boot. Default values range from 50 to 1200 ticks per second on real hardware. This is much too coarse-grained for ioprof.

5.2 processor-specific registers

For the high precision which is needed, ioprof now uses platform-dependent resources, a choice of precision over portability. Most modern processors include a counter register that is steadily incremented once at each clock cycle. This clock counter is the only reliable way to carry out high-resolution timekeeping tasks. The one big problem is of course that the register may even not exist for your platform, or it may be implemented in an external device and the register may or may not be readable from user space, it may or may not be writable, and it may be 64 or 32 bits wide. One of the most commonly available register is the timestamp counter (TSC), introduced in x86 processors with the Pentium and present in all CPU designs ever since including the x86_64 platform. It is a 64-bit register that counts CPU clock cycles and it can be read from both kernel space and user space. It therefore meets all requirements set by the LCG-environment. Via including the machine specific registers `asm/msr.h` one gains access to these registers through `rdtscl(low32,high32)`, `rdtscl(low32)` and `rdtscll(var64)`. On a 1GHz machine, the value `low32` returned by `rdtscl()` would overflow after 4.2 seconds and is therefore not usable but the 64bit value of `rdtscll()` would need more than 584 years - which should be sufficient for quite a time with even faster processors to come. It might be possible that Itanium processors are to be used in LCG. For this reason compatibility to ia64 code was assured by implementing a few lines of inline assembler code which read timestamp counter registers available on an itanium.

6 Fortran

A question was whether Fortran code could be used together with the library. A simple test-programm was written to determine whether Fortran is linked against the `libc`: The file was named `hw.F` and was compiled with intel's fortran compiler: This showed that fortran is in fact linked against the `libc`. A real world application named `CKMFitter` (see ?? on page ??) finally showed that fortran applications can be monitored as well using the ioprof-library.

```
1 implicit none
2 write (*,*) "Hello_world!"
3 STOP
```

Listing 6: Fortran example

```
1 bash: ifort -FR -o hw fortranTest.f
2
3 bash: ldd hw
4     linux-gate.so.1
5     libimf.so.6 => /opt/intel/fc/9.1.036/lib/libimf.so.6
6     libm.so.6.1 => /lib/tls/libm.so.6.1
7     libunwind.so.6 => /opt/intel/fc/9.1.036/lib/libunwind.so.6
8     libc.so.6.1 => /lib/tls/libc.so.6.1
9     libgcc_s.so.1 => /lib/libgcc_s.so.1
10    libdl.so.2 => /lib/libdl.so.2
11    /lib/ld-linux-ia64.so.2
```

Listing 7: compile Fortran example

7 Analysis of the gathered data

The first Step is to generate the log-files. Now they must be examined and parsed according to certain criteria. This can be a very complicated task involving many log-files from different processes running concurrently. A algorithm is needed for sorting all events according to their timestamps. But because of the size of these log-files it is not possible to read the log-files all at once. There are several solutions available. The first one was written in C to test whether the syntax and the data contained in the log-files is sufficient to generate the desired reports.

7.1 FileprofParser

FileprofParser is used to parse a single log-file. It was mainly written as a prove of concept to show that the log-file-format really works. Therefore this implementation only takes a single file as input to easy the development effort needed. The algorithm developed here was later re-implemented by the other analysers. Therefore, the following details and graphics apply to all analysers.

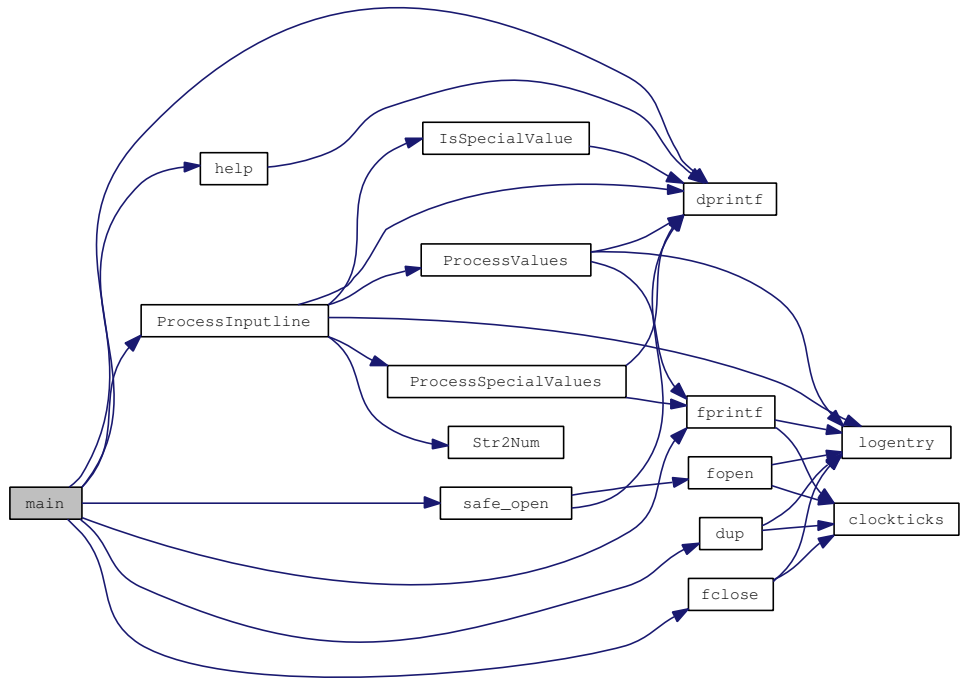


Figure 7: call-graph for the main() function of the FileprofParser

The functions help(), dprintf() as well as safe_open() and Str2Num() are little helpers and will not be explained further as their names are selfexplaining. As can be guessed from the function-names, the log-file is parsed one line at a time - this is done by ProcessInputline().

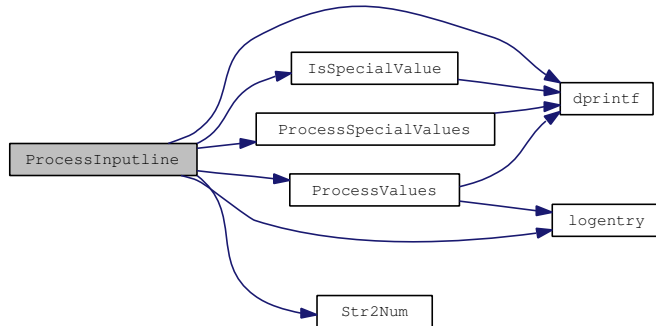


Figure 8: this function is called for each line in the logfile

The first thing to do here is to read the actual line from the logfile, and to disassemble it into its parts which are separated by a “;”. Then the function IsSpecialValue checks whether the actual line belongs to the header/footer (please see 7 on page 7) of the file or if it is a standard logentry. Depending on that either ProcessSpecialValues() or ProcessValues() is called.

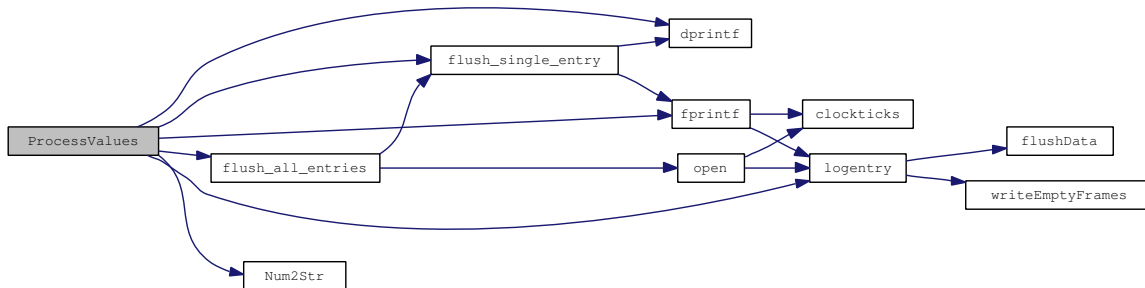


Figure 9: this function is called whenever a logentry is found

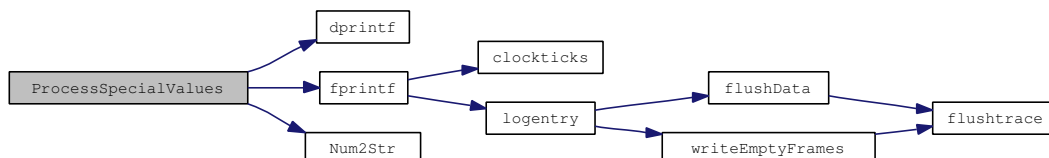


Figure 10: each line appertaining to the header or the “endofflogfile” line are special values

Whenever a interval is finished, all the data for the interval are gathered. All files having been accessed in that interval are written to the output-file as well as the tripple used by gnuplot.

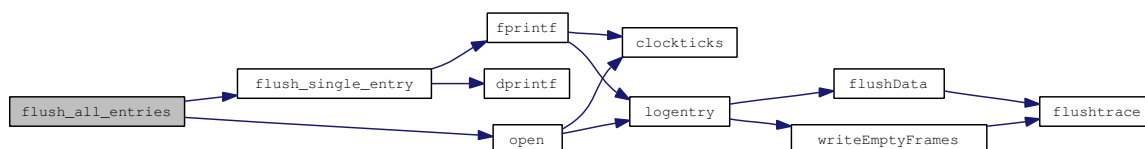


Figure 11: gather all data, write it to disk and flush all variables

As a result of the parsing a png-picture is generated with the help of gnuplot. It shows the accumulated in/out data per interval. The interval-size has to be provided at the commandline, otherwise on a defaultvalue of 1 second. The generated outputfile contains all the files accessed in the intervalls - these lines are simply ignored by gnuplot. At the end of each interval, a tripple is written, consisting of the interval-count, the bytes read and the written bytes. These three values are used to create the image. FileprofParser is a fast tool to evaluate the general I/O behaviour of a single process.

2. lcg-mon parsing during job-execution -same intervals for io and the rest -the monitoring log-files must be small enough to be parsed between 2 intervals, therefore not everything can be logged
3. lcg-mon parsing after jobexecution -log-files can be of arbitrary size -parsing may take very long - but parsed data is more accurate