From detailed analysis of IO pattern of the HEP applications to benchmark of new storage solutions

Jiří Horký†, Roberto Santinelli‡

†Institute of Physics, AS CR, Na Slovance 22, 182 21, Prague, Czech Republic
‡European Organization for Nuclear Research (CERN), Genève 23, Switzerland
E-mail: horky@fzu.cz

Abstract. The problem of file access on the site level has been one of the main issues since the start of the WLCG project. A lot of studies have already been performed using both industry standard benchmarks and actual physicist jobs. However, such studies are typically bound to one particular LHC experiment supported on a given site and/or one type of job. In this paper, we present an application suitable for detailed study of applications’ file access behavior. We have also developed an application for exact replaying of IO requests of the previously run applications. This enables to benchmark performance of solutions without a need of installing the whole working environment.

1. Motivation

Understanding of application’s behavior is essential for being able to improve its performance. In particular, when speaking of a file access, one is typically interested in a disk access pattern of the application, i.e. whether it reads data sequentially or randomly, how big typical IO requests are, how many of these requests the application does and how many files it opens per run. Once having this understanding one can very accurately explain why the application performs better on one file system than on another which is often a very difficult task. This information can be then exploited to better tune the environment. Section 2 deals with this topic.

On the other hand, when benchmarking a storage system, one usually wants to know how it behaves under the real load of his application. However, this is often hard to achieve because of the cost and complexity involved in setting up real applications. For instance, execution of an analysis or a reconstruction job of an experiment requires complex setup of software and access to the data files. For example, for ATLAS, one needs the right versions of gcc, python, ROOT[1], Athena (ATLAS analysis software) and several others depending libraries. Sections 3 and 4 deal with this topic.

We present a tool for IO profiling application called IOprofiler that can be used to determine all these required information.

We also present a tool that uses trace-and-replay mechanism for benchmarking storage solutions that minimize the cost of setting up the application environment while performing very similar as the original application. Both tools are freely available at [2].

Presented work is a part of the main author’s Master Science Thesis [3] which includes more technical details, detailed plots of IO patterns of different LHC experiments’ applications and their versions as well as benchmarks of various distributed file systems using the presented tool.
2. IO Profiling

The idea behind profiling is to catch all IO calls and inspect their count, offset within a file and duration. One can then reconstruct when and which parts of the file were read, sequentially or randomly.

2.1. IO profiling method

There are many possible methods of how to track the IO calls, differing on the layer they operate at (completely in kernel, user-space or combinations of the two), the complexity of its usage, ability to track hidden IO calls (caused by page faults of memory mapped files) and they also vary in a performance overhead they introduce.

We have evaluated several methods in detail, namely SystemTap[4], library preloading through LD_PRELOAD environment variable (the PlasticFS [5] and IOProf[6] application use this approach) and by exploiting ptrace(2) system call that is used in strace [7].

Taking into account advantages and drawbacks of the various methods, we decided to use the strace method, mainly because of its wide availability, no need to modify nor the application neither the running system (kernel) and ease of use when obtaining profiling information even without root privileges. Besides having the expected problem with memory-mapped IO calls, it can have a significant performance overhead of more than 100% of running time depending on the number of calls traced. It was measured that approximately 20 thousands of syscalls per second cause 1% of overhead on Intel Xeon E5520 2.27GHz which is reasonable for tracing LHC experiments’ applications reading ordered ROOT files.

The overhead problem that might affect exact timing of individual IO calls is not considered as a big issue, as we were mainly interested in the access pattern of the applications, which should not be affected. All the LHC experiments’ applications can be configured to use POSIX-like method for accessing data to enable us spotting all issued IO system calls.

2.2. IOprofiler

Because of the shortage of options for IO analyzing of a strace output (the only script that can be found concerning strace outputs and IO analysis is strace_analyzer [8] that is rather simplistic), a new tool called IOprofiler has been developed.

The goal was to provide fast and user friendly way of analyzing IO from a strace output. The idea was to provide a GUI application from which users could generate access pattern diagrams (individual read/writes plotted with file offset and call time as axes) of individual files, with some statistics in a few clicks. Because the strace output files produced by long running jobs that process tens of gigabytes of data could be very big (up to few gigabytes), there was also a need for an option to convert the text file into a more suitable architecture-independent binary format, that would be smaller and faster to read, as the text would not have to be parsed.

Because of a low level nature of system calls and requirement of fast processing, pure C was chosen as the programming language for the backend part, whereas the GUI part takes advantage of the Matplotlib library [9] and is therefore written in Python using PyQt framework.

2.3. File Descriptors Handling

The basic idea of using strace output is to simply go through all read, write and lseek system calls and plot them with file offset and time as the axes to see whether a file is accessed sequentially or not. It turns out that it is quite more complicated.

The issue is that every such system call operates with a file descriptor instead of a file name, so one has to keep track of the mapping of the file descriptor to the file name it is referring to, which is surprisingly not a trivial task. To not go into too many technical details, we do not present all the problems involved in this paper. Please refer to [3] for detailed explanation. A
lot of emphasis has been put in handling this file descriptors-to-file names mapping right and fast.

2.4. GUI

To enable a user-friendly way of analyzing traced data, a simple GUI written in PyQt framework has been developed. It consists of two screens. The first one (figure 1a) shows a list of files read or written by a traced job with basic statistics for each file (number of reads, summary of read requests, summary of duration of requests). The list is sortable and the shown files can be filtered by using regular expression filters in the bottom of the screen.

The detailed information could be displayed for each file. This triggers a detail dialog with the list of all operations on the left side and the plot on the right side. There are three types of plots. The default one, the pattern diagram, shows individual requests performed with a file.
offset as x axis and time elapsed from opening the file as y axis as shown in figure 1b (IO pattern of a real life LHCb analysis job). The other plots are histograms of requests’ size, duration and speed.

Taking advantage of using Matplotlib, the program enable users to zoom the plots and highlight individual request by either clicking in the pattern diagram, or selecting the requests from the list. The plots could be exported to both vector and scalar graphic formats.

Although rather simple, the GUI makes really fast and efficient analysis of a job’s access pattern possible.

3. Benchmarking

Benchmarking a storage system is a complex task. There are three approaches one might adopt, based on the trade-off between complexity of a benchmark setup and fidelity of it. a) Run the real application on the test system and measure some application metrics; b) Collect traces from a running application and replay them back on to the storage systems to be evaluated; or c) Use synthetic benchmarks to generate workloads and measure the IO system performance for different parameters of the synthetic workload.

While the first method is ideal as it measures the performance of the system at the point that is the most interesting for actual users, it is also the most difficult to set up because of the cost and complexity involved in setting up real applications. Moreover, this approach is not always possible.

Trace and replay is particularly attractive as it eliminates the need of understanding the application in detail. The question is the validity of the abstraction, especially the validity of the trace in a modified system.

An application called IOreplay has been developed for benchmarking using the trace and replay paradigm and it is presented in this section including a validity study.

3.1. Replaying

The idea of trace and replay method is to perform every IO operation that was previously recorded by the strace program. Doing so, the obtained results should be very similar to running the actual job as all requests that hit the hard drive are performed in the exactly same order with the same size. The main difference is in the timing of the calls as strace itself has quite a big overhead. As described in [10], special care must be taken to overcome this issue.

It is important not only to replay the read and write requests, but also the metadata operations like stat and access system calls that are also very common and can represent a non-trivial part of the IO operations especially for distributed file systems with a remote metadata server.

4. IOreplay

An application that is able to replay every IO-related system call named IOreplay has been written in the C programming language. Only system calls that hit the hard drive are replayed.

It was decided to make the application single-threaded mainly because of the complexity of debugging multi-threaded programs and because of the fact that the physical applications that were straced are strictly single-threaded even though they involve more than one process.

The following sections describe the whole process of replaying in more details.

4.1. Preparing the Environment

In order to reliably replay all operations and to obtain the same results as in the original run of an application, it is important that the environment in which we replay is the same as it was when we recorded the run. In other words, every open, unlink, stat and any other system calls that originally failed should now fail as well, while the calls that succeeded should succeed.
For that purpose, two mechanisms were developed. The first one provides a means how to define files that should be ignored during replaying (no IO operations are done to those files) and also enables mapping between original files and files in a new environment that comes in hand when replaying on a different server with different mount points. The second mechanism efficiently checks for problems that would occur during replaying and suggests changes, which is useful to ensure that the environment is valid prior to benchmarking.

4.2. Timing Modes
As stated in [10], it is crucial to keep the delivery of the calls in as exact timing as in the original application. Skew of more than few hundred microseconds causes significant difference in IO planning. To keep delivery of the IO calls as accurate as possible, we use periodical CPU instruction counter reading (representing the count of CPU cycles done since reset) with spinning in a loop between the reads which is very accurate yet very fast way how to determine time. However, this approach has several drawbacks ranging from limited portability to problems on multi core CPUs that require special care.

The application provides three modes in which it can replay the calls:

- **AFAP mode** - The calls are performed as fast as possible, ignoring any delays caused by processing the data in the original run.
- **Exact mode** - The calls are made as close as possible to the time they were previously made (measuring from the start of the application). Spinning in a loop is performed until it is time to make the call.
- **Diff mode** - The IO calls are made with the same delays between them as in the original run. This mode usually gives the most reasonable results as it holds all think-times (e.g. data processing) of the original application and the only difference in total execution time is caused by different duration of the IO system calls. Moreover, the application allows to specify a multiplier by which these think-times should be shrunk/widen. This is handy for simulating difference in CPU speeds, when the ratio between the reference system and the tested one is known.

4.3. Scaling
In order to mark results obtained by replaying as valid, it must be shown that it behaves similarly to the original job even on different hardware and under different load of the system. In other words, that it scales well.

An ATLAS job that read ordered files created with ROOT 5.27/05 was used to confirm the scaling of the replaying. Every instance processed 9GB of its own copy of data to avoid influencing the page cache of the operation system that was also flushed after each run. The machine used for the comparison study was equipped with two Intel Xeon E5520 CPU (8 cores total), 16GB RAM and one 300GB SAS drive of 15k RPM on which the input data were stored. The diff timing mode was used in order to simulate processing of the data by the application.

The results are shown in figure 2. The figure shows the sum of time (with standard deviation) spent to run different number of concurrent jobs. The numbers are roughly the same up to nine of concurrent jobs. This was expected because of the way the IOreplay works and number of CPU cores in the machine. When overloading the node with more than eight jobs, the bottleneck starts to be the data processing (CPU). Contrariwise, the IOreplay still tries to keep the delays the calls, so it just does less cycles in the CPU spinning and thus runs faster than the original application.

The study clearly shows that the IOreplay can scale within a few percent of difference in comparison with the original application, provided the recording of the traces had reasonable overhead and that we do not overload the node.
5. Conclusion
We have introduced a tool for IO profiling, the IOprofiler. It enables fast and easy-to-use analysis of IO behavior of applications. We have also described the IOreplay, a benchmark using trace and replay mechanism, that can be used to evaluate performance of different storage solutions without a need to complexly setup the environment for real applications. This could be particularly useful for tender evaluations. Most importantly, it was shown that it can be used for a very accurate benchmarking in comparison with respect to the original application. Both tools are freely available at [2].

Acknowledgments
This work was supported by Academy of Sciences of the Czech Republic and European Organization for Nuclear Research (CERN).

References
[1] Brun R and Rademakers F 1997 Nucl. Instrum. Meth. A389 81–86
[2] Horky J 2010 IOapps toolkit - IOprofiler and IOreplay tools http://code.google.com/p/ioapps/
[3] Horky J 2010 Improving Efficiency of HEP applications, Msc. Thesis http://www.farm.particle.cz/Msc.
[4] Thesis.Jiri.Horky.pdf
[5] 2010 Systemtap: tracing/probing tool for complex tasks http://sourceware.org/systemtap/
[6] Plastic File System http://plastics.sourceforge.net/
[7] William T 2010 IOProf http://www.benchit.org/wiki/index.php/IOprof
[8] 2010 Strace, system call trace tool http://sourceforge.net/projects/strace/
[9] 2010 Strace analyzer, IO Profiling of Applications http://clusterbuffer.wetpaint.com/page/Strace+
Analyzer++Next+Generation
[10] 2010 matplotlib, a python 2D plotting library http://matplotlib.sourceforge.net/
[11] Anderson E, Kallahalla M, Uysal M and Swaminathan R 2004 FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies (Berkeley, CA, USA: USENIX Association) pp 45–58