Pianola: A script-based I/O benchmark

John May
Lawrence Livermore National Laboratory
Email: johnmay@llnl.gov

Abstract

Script-based I/O benchmarks record the I/O behavior of applications by using an instrumentation library to trace I/O events and their timing. A replay engine can then reproduce these events from the script in the absence of the original application. This type of benchmark reproduces real-world I/O workloads without the need to distribute, build, or run complex applications. However, faithfully recreating the I/O behavior of the original application requires careful design in both the instrumentation library and the replay engine. This paper presents the Pianola script-based benchmarking system, which includes an accurate and unobtrusive instrumentation system and a simple-to-use replay engine, along with some additional utility programs to manage the creation and replay of scripts. We show that for some sample applications, Pianola reproduces the qualitative features of the I/O behavior. Moreover, the overall replay time and the cumulative read and write times are usually within 10% of the values measured for the original applications.

1. Introduction

I/O benchmarks help system designers evaluate the performance of file systems and storage devices. Two types are commonly used: *Application-based I/O benchmarks* measure the system response to a specific workload. They produce relevant data, but they can be difficult for nonspecialists to build and run. Moreover, they may not be distributed widely if they reveal sensitive algorithms or data. *Synthetic benchmarks* measure overall I/O system performance using standard or customized I/O access patterns. They are usually easier to build and run than application-based benchmarks, but they may not accurately replicate specific workloads of interest.

A third type of benchmark, which we call *script-based,* combines the realism of application code with the convenience of synthetic benchmarks. In this model, an instrumentation library records I/O events and timing data from a running application to produce a script. A replay engine reads this script (or a processed version of it) and reproduces the I/O events at the proper time intervals. (See Figure 1.) The replay engine can be simple to compile and use, and scripts can be generated from real applications without revealing sensitive information.

Others have experimented with script-based I/O benchmarks (for example, Mesnier et al. [1]). Extracting accurate and appropriate I/O information from an application is challenging, and so is replaying the I/O events at the correct time intervals. This paper describes our efforts on both fronts. Our main contributions are:

- Determining the right level in the software stack at which to monitor I/O events;
- Using binary instrumentation to record these events with minimal overhead;
- Compressing and preparsing scripts to minimize the impact on the I/O system of reading a script during replay; and
- Implementing a replay engine that can accurately reproduce the time intervals between I/O events.

Our system is called *Pianola.* (The name is taken from an early model of player piano.) Pianola currently replicates the overall execution time and the I/O time of tested applications within about 10%.

2. Tracing I/O events

The central question in designing a system to record I/O events is: Which events should be recorded? A simple approach would be to record each call that an application issues to an I/O-related function in the standard Unix library. For example, if a C program calls **fprintf**, record the **fprintf** call, and if a Fortran program calls an intermediate library that in turn calls the Unix **write** function, record the call to **write**. This approach makes it easy to instrument applications

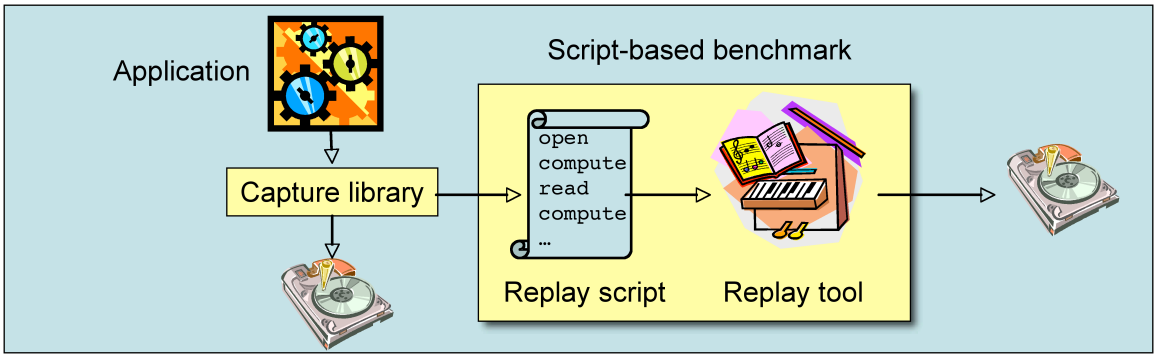

Figure 1. A script-based I/O benchmark records the I/O events that an application generates, along with the time intervals between then. The resulting script can be fed to a replay engine, which reproduces the I/O behavior of the original application.

without recompiling or relinking them, using library interposition (at least in many Unix implementations). Linux, for example, has a dynamic loader that lets users substitute their own versions of library functions for the standard ones. The customized versions can record information about a function call and then call the standard version of the function to complete the operation that the user requested.

Instrumenting an application's calls to the high-level I/O functions has two problems. First, a significant amount of computation may occur within a high-level function before (or after) any data is moved between the application and the I/O system. This computation should not be counted in assessments of I/O system performance because it depends entirely on the processing power and memory performance of the system, and not on the performance of the storage hardware or its interconnection network. This is not to say that the computation time that elapses during a high-level I/O call should not be measured and reproduced in a benchmark, only that it should be counted in the compute time and not the I/O time.

The second reason for not instrumenting high-level I/O calls is that they often handle data in small increments. An application may call **fprintf** many times before an internal buffer fills up and the library issues the **write** system call to move the data to storage. Recording the many **fprintf** calls instead of the single **write** call increases the number of events logged. As we will show later, minimizing the size of the log improves the fidelity of the replay.

A better approach is to log only the calls to system-level functions, since these are at the interface between the application and the I/O system. However logging these calls presents a new challenge: library interposition does not allow us to intercept system calls that high-level I/O functions make. For example, an application's direct call to **open** can be captured, but a call that **fopen** makes to **open** cannot be captured using standard interposition techniques. In fact, **fopen** may not call the **open** function at all; instead it may issue an equivalent series of low-level instructions that cause the system to open the file.

We initially considered two possible solutions, both of which proved unworkable. Since the source code for the GNU/Linux standard I/O library is available, we could modify it to add our instrumentation at the appropriate level and then use library interposition to link our instrumented version of the library dynamically to applications. However, this code is quite complex, so finding all the locations where high-level functions make system calls is difficult and error-prone. Moreover, we would have to reimplement the instrumentation in each version of the library we wished to support. The second possibility was to use the Linux strace utility to intercept the I/O system calls using the system's debugger interface. Strace is simple to use and provides useful and detailed information, but in our tests it incurred tens of microseconds of overhead for each monitored I/O event. The resulting timing data exaggerates the duration of the computation intervals between the I/O events. As a result, it was not possible to produce an accurate script from this data.

The solution we settled on uses binary instrumentation. This technique has been available for many years in tools such as DynInst [2] and DPCL [3]. The idea is to scan the executable code of an application after it has already been loaded in memory, looking for patterns of instructions that form events of interest, such as calls to a particular function or system calls that invoke kernel operations. The binary instrumentation tool replaces these instructions with a call to code that

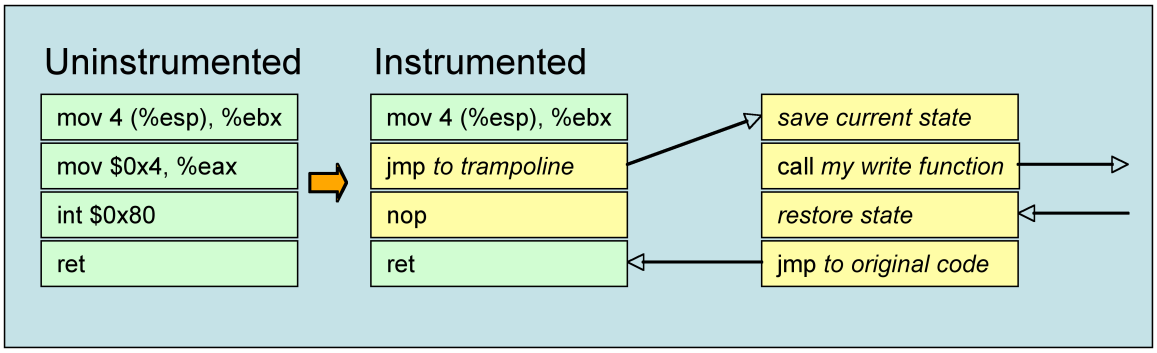

Figure 2. Using the Jockey dynamic instrumentation library, we can modify the executable code of an application after it has been loaded into memory. Each time Jockey identifies an I/O system call in the uninstrumented code, it replaces the relevant instructions with a call to some "trampoline code" that saves the current state and then calls an instrumented version of the I/O operation. When that version completes, the trampoline code restores the state and jumps back to the original code.

does some other operation chosen by the tool designer, such as recording the event or examining variables. The inserted code may then complete the action that the original instructions implemented and jump back to the instruction that follows the ones that were replaced. The scanning and replacement may be done by an external process (with appropriate access permission), or it may be done by a library that the executable itself loads.

To apply this general technique to our application, we have chosen the Jockey library [4], which is designed specifically to instrument system calls on Linux platforms. Applications can load Jockey dynamically using the Linux library interposition facility.

In Linux systems, the pattern for system calls consists of an instruction that places a system call identifier in a particular register, following by an **interrupt** instruction with the hexadecimal value 0x80. When Jockey finds these instructions, it replaces them with instructions to jump to a specially-prepared region of "trampoline code" that saves the current processor state and then calls a user-defined function that substitutes for the original system call. In our case, this function records the system call parameters and timing, and then issues the original system call. When the user-defined function returns to the trampoline code, the trampoline code restores the processor state and jumps back to the instruction that followed the system-call pattern. Figure 2 illustrates the process.

Jockey scans and modifies each instance of an I/O system call only once per execution of the program. Whenever the program encounters a modified I/O call, it simply executes the substituted instructions that call the instrumented code. As a result, a dynamically instrumented application can execute its instrumentation as efficiently as if it had been compiled into the original application. To see how the instrumentation overhead differs between Jockey and strace, we measured the compute (non-I/O) time for one of our sample applications with Jockey and with strace. For comparison, we also measured the user time of the uninstrumented application with the Unix time utility. For the Jockey-instrumented run, the average compute time over three trials was 289 seconds; for time it was 284 seconds, indicating for this test that our binary instrumentation added 5 seconds to the measured compute time. By contrast, strace measured the 347 seconds of compute time, 63 seconds more that the time utility.

Much of the rest of our instrumentation library is adapted from the //TRACE system [1]. Although //TRACE was developed to instrument I/O calls in parallel applications, our focus has been on low-overhead instrumentation of sequential programs. Nevertheless, we have borrowed the //TRACE approach of setting up a daemon process separate from the main application to collect and store trace data.

3. Processing the script

The Pianola instrumentation library outputs a text-formatted script similar to the output of strace. Each line in the script represents an I/O event. A sample of typical output appears in Figure 3.

The first item in each line is the time in seconds since the beginning of the preceding event; next comes the event with its parameters and return value; finally, the measured duration of the event, also in seconds, appears in angle brackets.

```
0.00010284 openm("/home/john/somefile.txt", 548, 384) = 7 <0.00012080>
0.00014123 openm("/home/john/otherfile.txt", 626, 436) = 8 <0.00003775>
0.00004879 close(8) = 0 <0.00002225>
0.00002949 open("/home/john/otherfile.txt", 516) = 8 <0.00002033>
0.00003065 llseek(8, 2147479512, 2147479512, 0) = 0 <0.00000561>
0.00001730 write(8, -1078271740, 24) = 24 <0.00009151>
```

Figure 3. Initial output from Pianola is similar to strace output.

```
openm ( pianola_tmp1 548 384 ) = 7
compute ( 0.000020 )
openm ( pianola_tmp2 626 436 ) = 8
compute ( 0.000011 )
close ( 8 ) = 0
compute ( 0.000007 )
open ( pianola_tmp2 516 ) = 8
compute ( 0.000010 )
_llseek ( 8 2147479512 0 ) = 2147479512
compute ( 0.000012 )
write ( 8 24 ) = 24
compute ( 0.000033 )
```

Figure 4. Intermediate script corresponding to Figure 3 has substituted file names and **compute** commands to specify interevent timing.

This format has the advantage of being both human-readable and portable among machines. The latter characteristic is especially important, since replay scripts must be usable on a wide range of platforms. In the example above, the **open** and **openm** calls specify flags as numeric values. (The two forms of **open** distinguish calls with and without the optional mode argument.) Since the values of flags vary across systems, the instrumentation library converts **open** flags to canonical values before writing them in the script. Later platform-specific processing converts the flags back to system-dependent values. (Strace spells out the names of flags, for example, O_CREAT | O_RDONLY.)

A potential problem with trying to replay **open** events on another machine is that the target system may not have the same directory hierarchy. If the directory "/home/john doesn't exist on a target machine, the **open** call will fail. To address this problem, we have written a simple utility that examines each **open** call in a script. If the file is being created, the utility substitutes a temporary name with no directory prefix. Since the replay engine will write meaningless data to this file, giving it a different name helps avoid confusion with the output files of the original application. If the file is not being created, then the name is used as is, but any directory prefix is stripped. However, if the file is in a standard directory, such as "/usr/include" or "/proc", then it is left unchanged. The utility outputs a new version of the script with these substitutions and reports all the changes it made, so the user can arrange to have files with the correct names available when the script is run on a target machine. Of course, "standard" directories on one platform may not be standard on another, so this solution isn't perfect, and it may be necessary to edit filenames by hand in some cases.

The Pianola software further processes the script in several steps before it is ready to be replayed. First, it computes the intervals between each pair of successive events and generates a new script in which I/O events alternate with **compute** events that simply specify a delay time. At the same time, certain I/O events that do not involve files (such as output to the terminal or data transfer over sockets) are removed. The resulting *intermediate script* is still in plain-text format and is portable across platforms. Figure 4 shows the intermediate script corresponding to Figure 3, with the full path names replaced by temporary file names. Unlike the original version, the intermediate script does not contain any data on the duration of the I/O events that were recorded from the original application; only

the time *between* events appears. The time spent in the application's I/O operations is not needed for replay, and in fact it may be distorted because of the additional time spent in logging.

This translation step closely follows the model of //TRACE. The next section describes the remaining steps in processing and replaying a script.

4. Replaying events

The replay engine could parse and execute the intermediate script directly, but that has two drawbacks. First, parsing lines of text can take a significant amount of time. For I/O events that are closely spaced in time, the replay engine could fall behind schedule as it interpreted each line of the script. Second, the script itself may be large, and the act of reading it may compete for bandwidth and file system buffer space with the I/O events that the replay engine is reproducing.

To address the first problem, we parse the text-formatted script in advance into a binary format that the replay engine can interpret much more quickly. This translation makes the resulting script platform-dependent because of varying integer sizes and byte orders, so it must be done on the target system (or a compatible system). In the binary format, there is one record per event, and each record has fields for the event type, the parameters, and the return value. String arguments are stored in event order in a separate file. This allows the replay engine to read a series of events directly into an array of records, without trying to distinguish between events and strings.

We reduce the impact of reading the script on the overall I/O performance by writing the binary script data in compressed format. We simply use the open source zlib compression library to output the binary script. In our experiments, the uncompressed binary representation of the script data is similar in size to the text representation. This is because the fixed-format records have enough fields to hold all the parameters for any known I/O operation, and most I/O events do not require all the fields or use all the available precision. Compressing the binary data typically reduces it by a ratio of about 30:1 or more. The following table shows the compression ratios for the applications presented in Figures 6 and 7.

	Ingest	**Miranda**
Uncompressed	60 MB	260 MB
Compressed	1.5 MB	6.5 MB
Ratio	30:1	40:1

Figure 5 shows the processing sequence for the replay script. Of course, when the replay engine reads the compressed script, it must do some computation to get the uncompressed I/O events. However, we can defer this processing to periods of time when the engine would otherwise be idle. The replay engine attempts to read and decompress script data only during the **compute** intervals between the scripted I/O events. During these periods, it fills a buffer with decompressed events to be replayed, so events are ready to execute as soon as the compute interval expires. When an event requires a string argument, the replay engine reads it from the separate strings file (which is also compressed).

Instead of preparsing and then compressing the script, we could have simply compressed the text-formatted script and then decompressed and parsed it in the replay engine during the **compute** intervals. However, the text-formatted script compresses only by ratio of about 2:1, rather than the 30:1 ratio of the binary version, so this approach would increase the script's impact on the I/O system during replay.

We could also compress the text-formatted script as the instrumentation library outputs it, but there is little benefit to this. As noted earlier, the important time intervals are those between I/O events, not the timing of the events themselves. Reducing the perturbation of the I/O events while they are recorded does not improve the fidelity of the replay.

4.1. Memory footprint

In our testing of Pianola, we have hypothesized that the small memory footprint of the replay engine could cause it to interact with the I/O system differently from a larger-memory application. Perhaps it leaves more memory to the operating system for file buffering than the original application does. With more buffer space available, the operating system could complete more I/O operations without accessing the disk.

To test this theory, we modified our code to record **brk** and **mmap** system calls, which modify the memory allocation. The replay engine then allocates chunks of memory to simulate the effects of **brk** and **mmap**. During idle periods, the replay engine accesses selected addresses in these chunks to ensure that virtual memory is paged in. We access addresses with a 1024-byte stride so that we touch a large number individual blocks during the the idle period. Of course, this approach probably does not replicate the memory access pattern of the original application, but it does maximize the memory footprint of the replay engine. Nevertheless, our tests showed that although we could

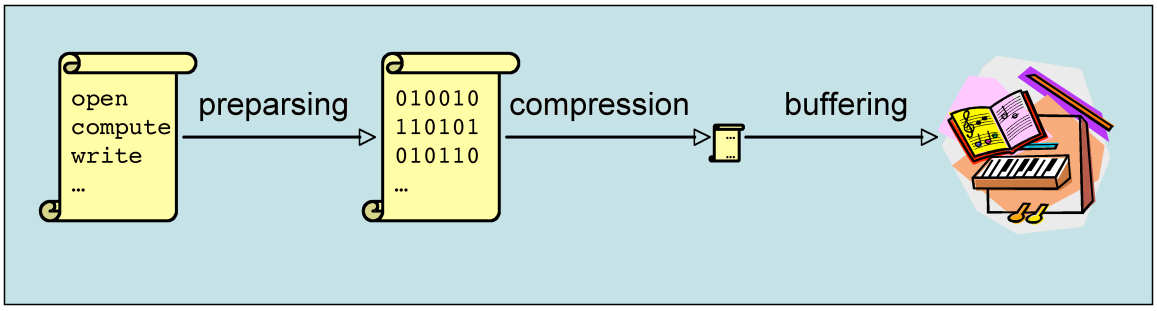

Figure 5. The plain-text script that the instrumentation library generates is parsed to binary format and then compressed. The replay engine reads the compressed script and decompresses sections of it into a buffer before executing the commands. Preparsing the data allows the replay engine to execute commands quickly, while compressing the script minimizes the impact on the I/O system of reading the script.

approximate the memory footprint of the application, there was no noticeable effect on the execution time of the I/O operations.

5. Evaluation

We generated replay scripts from several data-intensive applications and compared the I/O activity profiles of the replayed scripts with the original applications. Figures 6 and 7 compare the I/O profiles for two applications with the profiles for the replayed sequences of operations. One ingests a large graph dataeset and converts it to an internal format for further processing. The other is the Fortran I/O kernel of a hydrodynamics application called Miranda. The profiles plot the cumulative time spent in read and write operations against the cumulative execution time. Although the profiles do not overlap completely, their shapes are similar, and the I/O and total execution times are within about 10% in most cases. The following table summarizes the accuracy of replay for these benchmark runs. (All times are in seconds.)

	Ingest	**Miranda**
Application read	35.8	19.9
Replay read	35.7	13.8
Read time ratio	*0.997*	*1.44*
Application write	12.8	73.9
Replay write	12.5	70.9
Write time ratio	*0.977*	*0.959*
Application runtime	334	274
Replay runtime	319	250
Runtime ratio	*0.955*	*0.912*

The largest mismatch between the application and replay times was for Miranda's read operations. Interestingly, the Ingest benchmark also shows the replay's read events running faster than the application's early in the execution. (Refer again to Figure 6.) Later, the replay read time "catches up." Perhaps some read buffering mechanism is working more efficiently for the replay engine than for the application, but the benefits disappear as more data is read.

6. Conclusions and future work

Script-based benchmarks offer several advantages over application-based benchmarks and synthetic benchmarks, but they also have some disadvantages. First, a script captures the application's behavior only for a single execution, reflecting a particular set of input parameters and data. While the behavior of other benchmarks can be often modified though small input files or command-line arguments, replicating different scenarios with a script-based benchmark requires different scripts. Second, for each file the original application reads there must be a file of the same size (though not necessarily the same content) available for the replay engine to read. For applications that access many files, setting up the necessary environment on a target system could be tedious.

Generating an I/O benchmark from a script of recorded events is a straightforward concept, but accurately emulating the application's I/O behavior requires some care. The most important considerations we found are:

- Recording events at the right level in the software stack;
- Minimizing the overhead of the instrumentation so that interevent timing can be gathered accurately; and
- Minimizing the effect of reading the script on the I/O behavior of the replay engine.

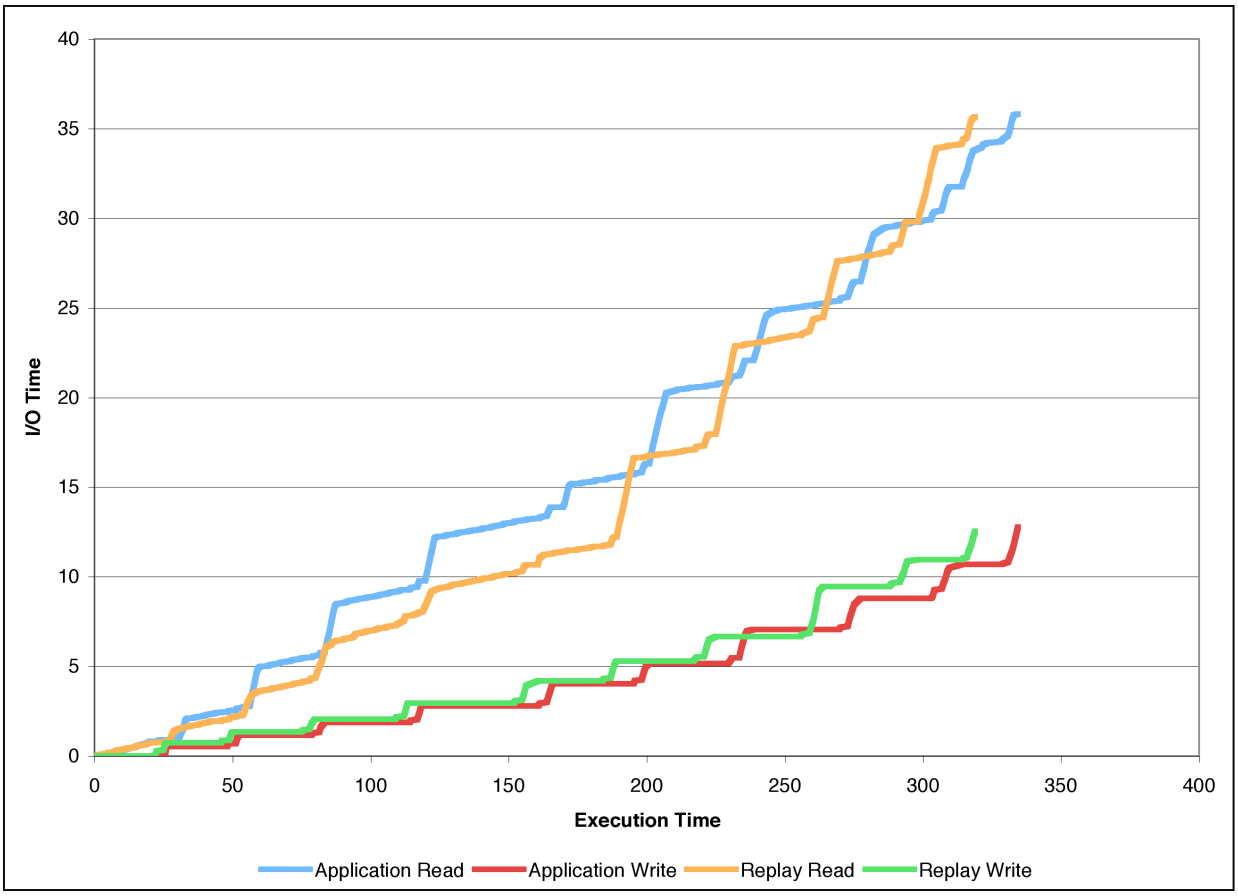

Figure 6. This I/O time profile shows the cumulative time spent in read and write operations versus the total execution time for a C-language graph ingestion application (blue and red lines) and the replay engine's recreation of the same I/O events (orange and green lines). Although the patterns do not line up exactly, the profiles are similar, and the I/O times and total execution times are within 10%.

Although the fidelity of Pianola's replay is acceptable for many purposes, there is probably room for improvement. It should also be possible to extend Pianola to work with parallel I/O systems. Much of the infrastructure for this conversion has already been developed in the //TRACE library.

Acknowledgements

Maya Gokhale gave advice and support throughout this project. Roger Pearce developed the software that plots the I/O profiles. The //TRACE team gave us a prerelease version of their code. We had helpful discussions with John Gyllenhaal, Mark Grondona, and Ben Woodard. This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-407121

References

[1] M. Mesnier, M. Wachs, R. R. Sambasivan, J. Lopez, J. Hendricks, and G. R. Ganger, "//TRACE: Parallel trace replay with approximate causal events," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, Feburary 2007.

[2] J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic program instrumentation for scalable performance tools," in *Proceedings of the Scalable High Performance Computing Conference (SHPCC)*, May 1994.

[3] L. DeRose, T. Hoover Jr., and J. K. Hollingsworth, "The Dynamic Probe Class Library—An infrastructure for developing instrumentation for performance tools," in *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, April 2001.

[4] Y. Saito, "Jockey: A user-space library for record-replay debugging," in *AADEBUG'05: Proceedings of the Sixth*

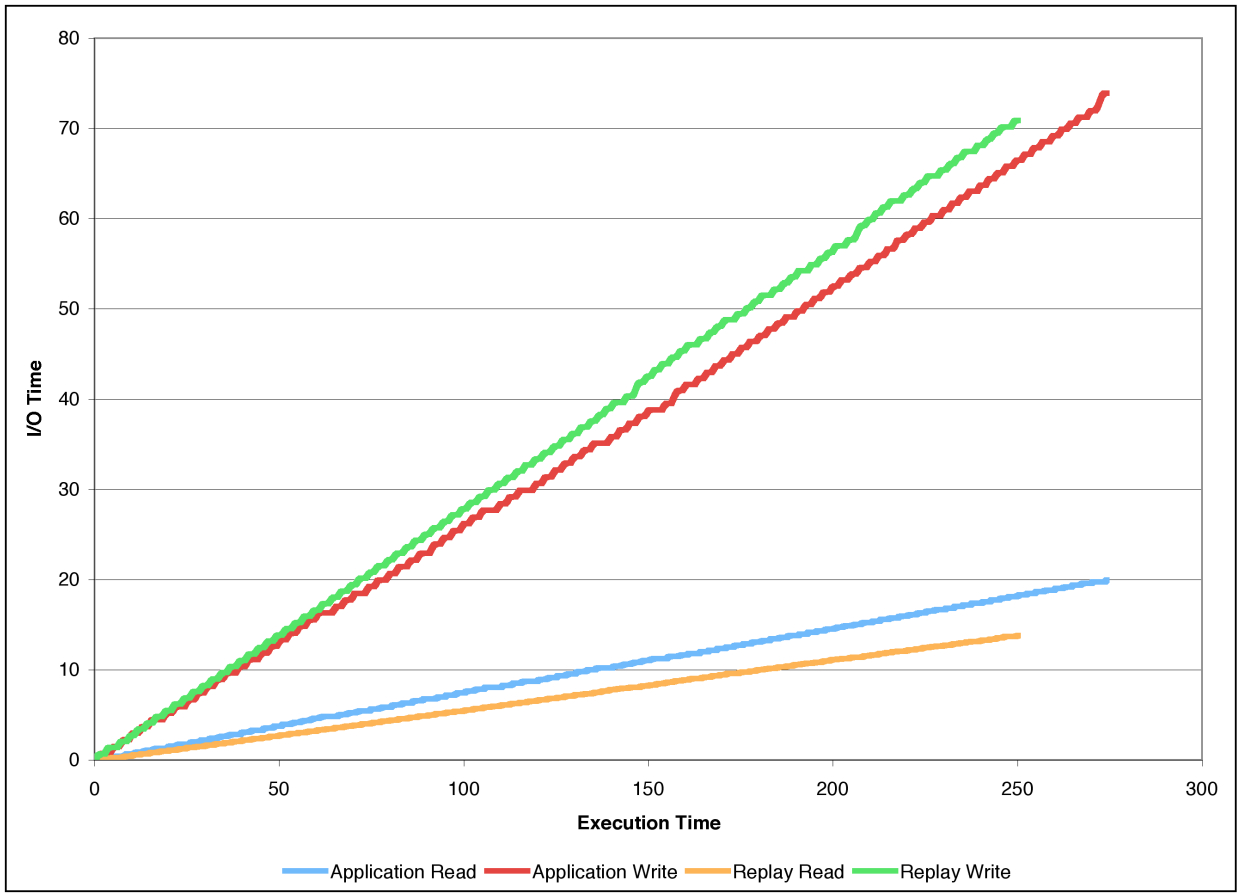

Figure 7. Similar to Figure 6, the profiles for the I/O kernel of a Fortran hydrodynamics code and the replayed version of the kernel show reasonably good agreement, although the replayed read operations finished somewhat faster than in the original.

International Symposium on Automated Analysis-Driven Debugging, 2005, pp. 69–76.