Lawrence Livermore National Laboratory

Pianola: A script-based I/O benchmark

John May

PSDW08, 17 November 2008

Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94551

This work performed under the auspices of the U.S. Department of Energy by
Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

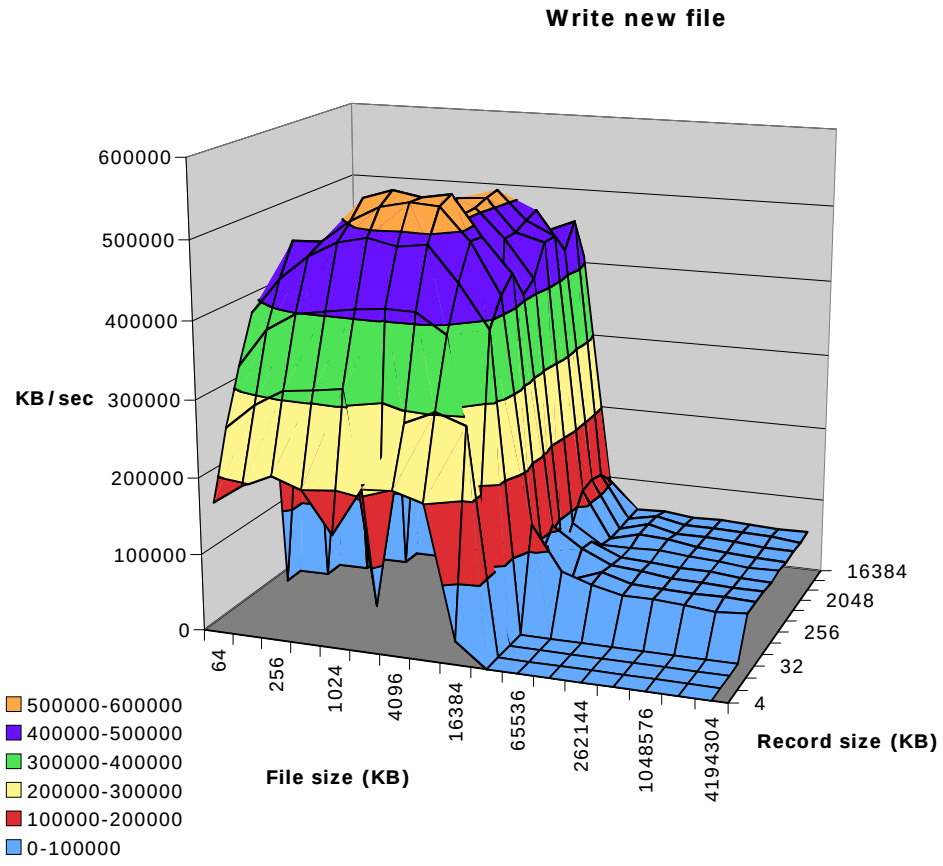
LLNL-PRES-406688

I/O benchmarking: What's going on here?

- Is my computer's I/O system "fast"?
- Is the I/O system keeping up with my application?
- Is the app using the I/O system effectively?
- What tools do I need to answer these questions?

- And what exactly do I mean by "I/O system" anyway?
  - For this talk, an I/O system is everything involved in storing data from the filesystem to the storage hardware

Existing tools can measure general or application-specific performance



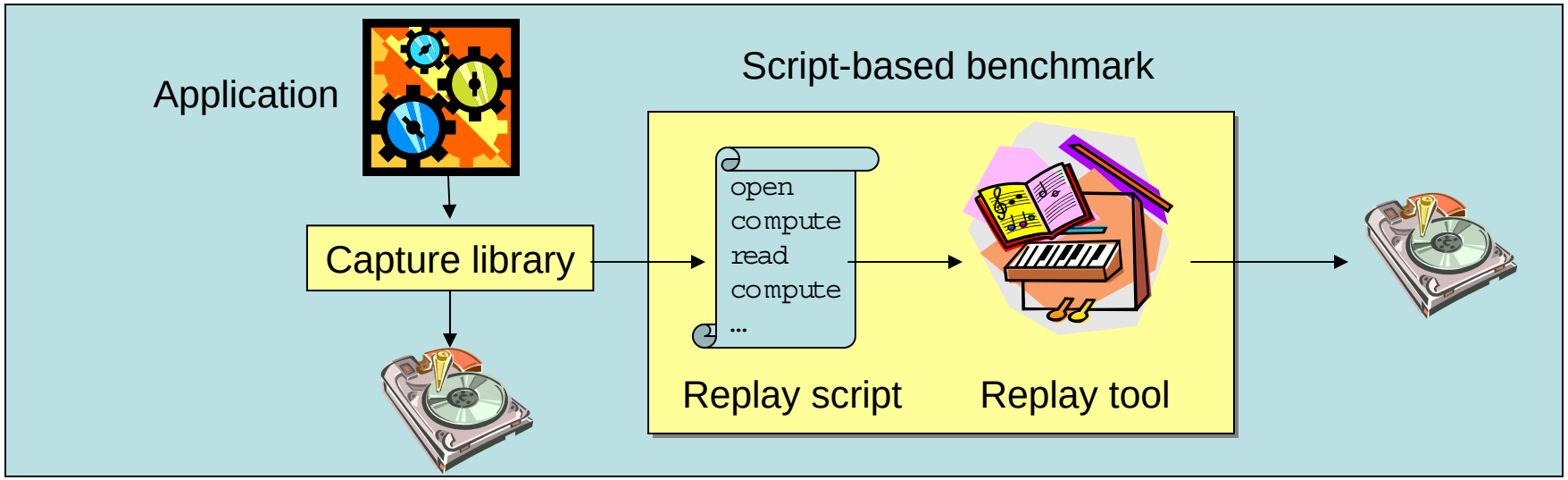
- IOzone automatically measures I/O system performance for different operations and parameters
- Relatively little ability to customize I/O requests

- Many application-oriented benchmarks
  - SWarp, MADbench2…
- Interface-specific benchmarks
  - IOR, //TRACE

Measuring the performance that matters

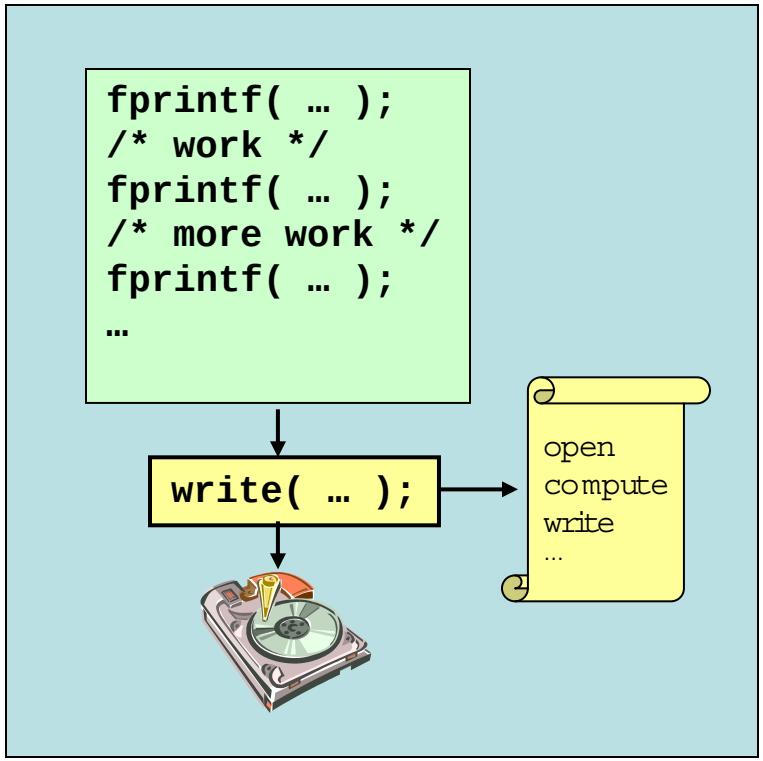
- System benchmarks only measure general response, not application-specific response
- Third-party application-based benchmarks may not generate the stimulus you care about
- In-house applications may not be practical benchmarks
  - Difficult for nonexperts to build and run
  - Nonpublic source cannot be distributed to vendors and collaborators
- Need benchmarks that…
  - Can be generated and used easily
  - Model application-specific characteristics

Script-based benchmarks emulate real apps



- Capture trace data from application and generate the same sequence of operations in a replay-benchmark
- We began with //TRACE from CMU (Ganger's group)
  - Records I/O events and intervening "compute" times
  - Focused on parallel I/O, but much of the infrastructure is useful for our *sequential* I/O work

Challenges for script-based benchmarking: Recording I/O calls at the the right level



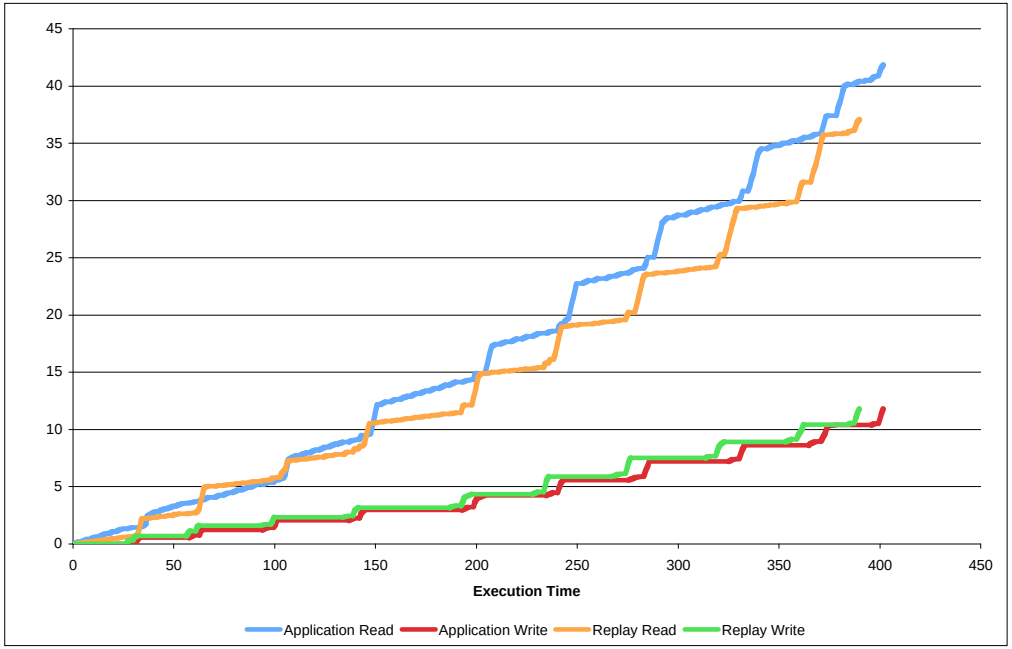
- Instrumenting at high level
  - + Easy with LD_PRELOAD
  - - Typically generates more events, so logs are bigger
  - - Need to replicate formatting
  - - Timing includes computation
- Instrumenting at low level
  - + Fewer types of calls to capture
  - + Instrumentation is at I/O system interface
  - - Cannot use LD_PRELOAD to intercept all calls

First attempt at capturing system calls: Linux strace utility

```
$ strace -r -T -s 0 -e trace=file,desc ls
     0.000000 execve("/bin/ls", [, ...], [/* 29 vars */]) = 0 <0.000237>
     0.000297 open("/etc/ld.so.cache", O_RDONLY) = 3 <0.000047>
     0.000257 fstat64(3, {st_mode=S_IFREG|0644, st_size=64677, ...}) = 0 <0.000033>
     0.000394 close(3)                  = 0 <0.000015>
     0.000230 open("/lib/librt.so.1", O_RDONLY) = 3 <0.000046>
     0.000289 read(3, ""..., 512)       = 512 <0.000028>
     ...
```

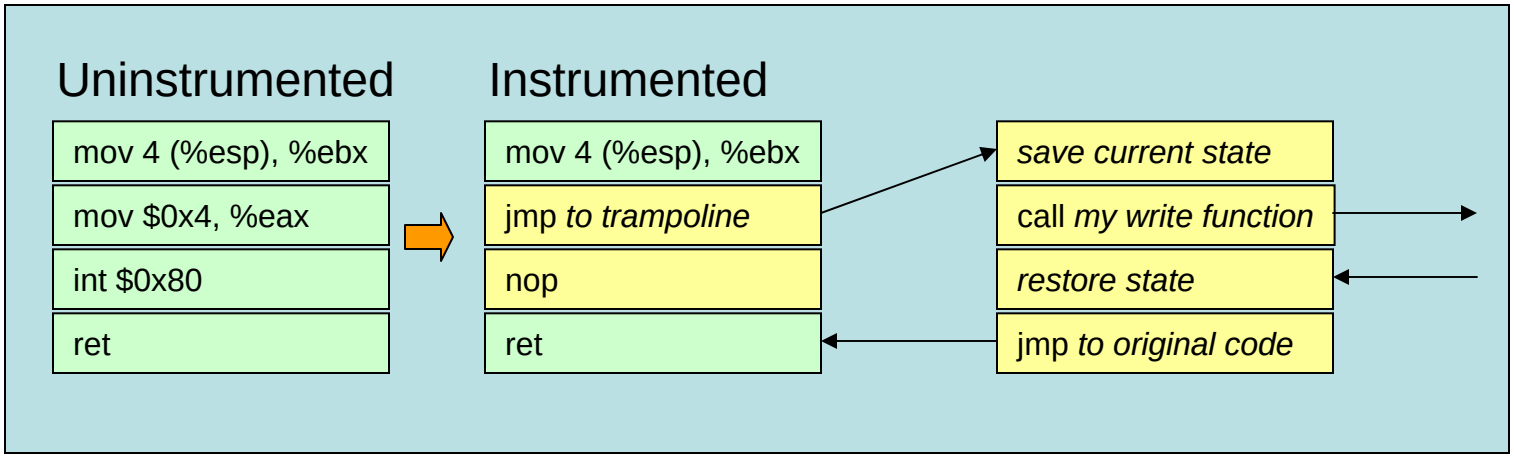
- Records any selected set of system calls
- Easy to use: just add to command line
- Produces easily parsed output

Strace results look reasonably accurate, but overall runtime is exaggerated



	Read (sec.)	Write (sec.)	Elapsed (sec.)
Uninstrumented	--	--	324
Instrumented Application	41.8	11.8	402
Replay	37.0	11.8	390

For accurate recording, gather I/O calls using binary instrumentation

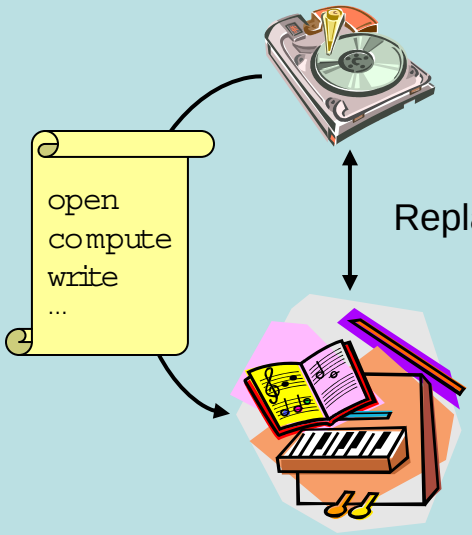


- Can intercept and instrument specific system-level calls
- Overhead of instrumentation is paid at program startup
- Existing Jockey library works well for x86_32, but not ported to other platforms
- Replay can be portable, though

Issues for accurate replay

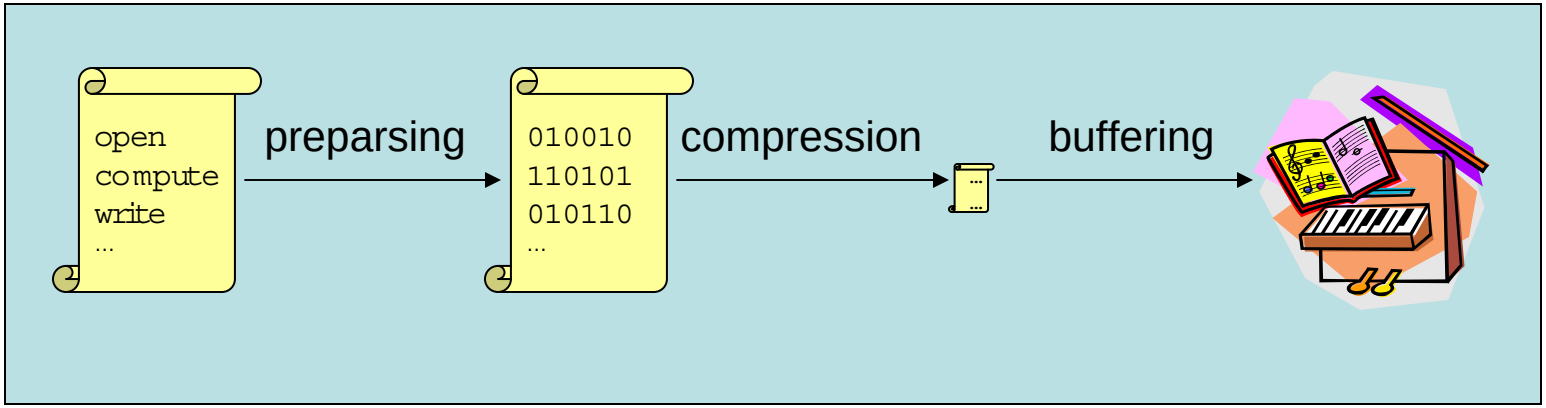
- Replay engine must be able to read and parse events quickly
- Reading script must not interfere significantly with I/O activities being replicated
- Script must be portable across platforms

Accurate replay
Minimize I/O impact of reading script



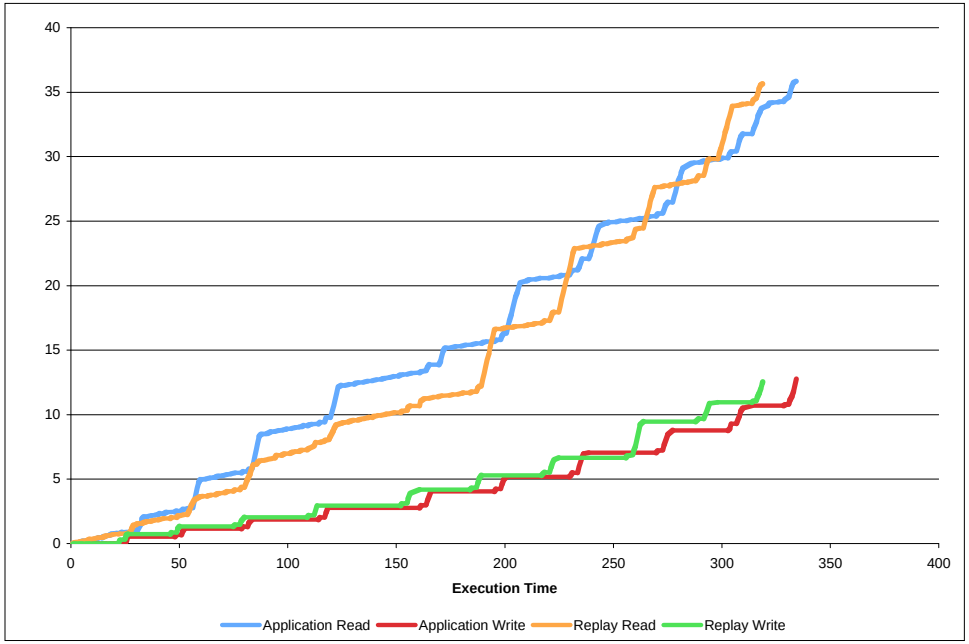
Correctly reproduce inter-event delays

Accurate replay: Preparsing, compression, and buffering



- Text-formatted output script is portable across platforms
- Instrumentation output is parsed into binary format and compressed (~30:1)
  - Conversion done on target platform
- Replay engine reads and buffers script data during “compute” phases between I/O events

Replay timing and profile match original application well



	Read (sec.)	Write (sec.)	Elapsed(sec.)
Uninstrumented	--	--	314
Instrumented Application	35.8	12.8	334
Replay	35.7	12.5	319

Things that didn't help

- Compressing text script as it's generated
  - Only 2:1 compression
  - Time of I/O events themselves are not what's very important during instrumentation phase
- Replicating the memory footprint
  - Memory used by application is taken from same pool as I/O buffer cache
  - Smaller application (like the replay engine) should go faster because more buffer space available
  - Replicated memory footprint by tracking brk() and mmap() calls, but it made no difference!

Conclusions on script-based I/O benchmarking

- Gathering accurate I/O traces is harder than it seems
  - Currently, no solution is both portable and efficient
- Replay is easier, but efficiency still matters
- Many possibilities for future work—which matter most?
  - File name transformation
  - Parallel trace and replay
  - More portable instrumentation
  - How to monitor mmap'd I/O?