fsdisco: a File System Behavior Discovery Tool

David Plonka
plonka@cs.wisc.edu
University of Wisconsin - Madison
Computer Sciences Department

February 15, 2007

Abstract

In this paper we present *fsdisco* - a portable perl script to discover the characteristics of a file system. We describe its design and report its capabilities and limitations, as currently implemented. We also present the results of an empirical study of the Linux ext2 file system. These results demonstrate that a portable perl script can perform the necessary fine-grained measures, allowing the programmer to conveniently maintain and reuse measurement code.

1 Introduction

Performance evaluation of computer systems can be challenging. To exercise system components and measure their performance, the programmer often resorts to writing, tweaking, and rewriting customized C or assembler code snippets. Furthermore, the necessary fine-grained time measurements might be performed using platform-specific methods, such as with the x86 `rdtsc` instruction. Unfortunately, this method is not reliable on systems with multiple processors, multicore processors, or variable-clock-rate (power-saving) processors[4], nor is it even available on other architectures. This leaves the programmer wanting a more convenient, portable way to obtain performance measures. To satisfy this desire, we developed *fsdisco*.

fsdisco is a perl script with modes and many options used to exercise a file system and expose its performance characteristics. It can `creat()` a file and subsequently perform repeated tests involving `write()`, `read()`, and `lseek()` operations with configurable byte-length, file offset/position, and direction. fsdisco can run commands before and after these tests to remount the pertinent file system and to gather related measurement information, such as memory buffer cache utilization. It also has options to post-process its measurements, such as determining median values, and can output them in a convenient format for further analysis or visualization.

2 Design

Clearly many time-critical sections of code must be written in C or assembler so that the don't incur the performance penalty of an interpreter. Perhaps surprisingly, it is possible to simply integrate C code into a perl script using the `Inline::C` module[3]. `Inline::C` enables one to embed C functions within a perl script, i.e. *in-line*, and to call them as easily as perl subroutines. While we won't expound on the clever mechanism by which this is accomplished, figure 1 is an example of perl script containing in-line C code to time a system call.

```
#! /usr/bin/perl
use Inline C;

my $minimum_estimate = gettimeofday_test(0);
if ($minimum_estimate >= .001) { # 1 millisecond
   die sprintf("minimum delta time too big: " .
        "%.06f seconds elapsed!\n", $minimum_estimate);
}
exit;

__END__
__C__
#include <stdio.h> /* sprintf */
#include <unistd.h> /* sleep */
#include <sys/time.h> /* for gettimeofday */

double
gettimeofday_test(unsigned int sleep_seconds) {
   struct timeval thentv, nowtv, elapsedtv;
   gettimeofday(&thentv, (void *)0);
   if (0 != sleep_seconds) {
      sleep(sleep_seconds);
   }
   gettimeofday(&nowtv, (void *)0);
   timeval_subtract(&elapsedtv, &nowtv, &thentv);
   return elapsedtv.tv_sec + elapsedtv.tv_usec/1000000.;
}
```

Figure 1: A perl script with in-lined C code

Using `Inline::C` thusly, a programmer can extend the perl interpreter to contain measurement functions of his choosing, and call them at will in his scripts.

fsdisco defers reporting output file I/O until after measurements are performed to minimize its affect on a system's file cache. There is a trade-off between poten-

tially affecting the measurements with a large measurement process versus affecting them by doing additional file I/O. As a compromise, fsdisco writes its results to files between test iterations to keep its process size smaller so as not to consume all physical memory or paging space. fsdisco has an option to summarize results from a set of output files; this is meant to be used after a set of test iterations have been completed.

Lastly, our tests generally make system calls frugally. This seems a sensible early optimization given that the behaviors under study almost certainly vary only when crossing kilobyte or larger boundaries. Performing lengthy sequences of single-byte writes or reads provides much more potentially noisy data points, but not more useful information.

2.1 Advantages

The advantages of performing measurements using perl, or another likewise-extensible scripting language, are primarily convenience and maintainability. With perl and `Inline::C`, the C code for measurements is compiled for you automatically on-the-fly when the script is executed, but only when that code changes. This makes it convenient to modify tests.

Initial analysis, such as calculating the median measurement values of test runs, can also be written in perl more conveniently than C, coupling the test and analysis into one self-contained script. The requisite file I/O to report measurements and analysis results can be done in perl. Lastly, fsdisco is portable; it is roughly as portable as perl itself. During fsdisco's development we tested it on the Linux and Apple Mac OS X operating systems on the x86 and PowerPC architectures, respectively.

2.2 Limitations

Because we have chosen to use only portable system calls, fsdisco's timer granularity is limited to that of the `gettimeofday()` call on the system under test. Before performing measurements, fsdisco attempts to determine `gettimeofday()`'s granularity. If it is not able to measure an interval on the order of microseconds, it report an error and exits. This could happen due to implementation restrictions but may also be due to system load, so it's advisable to run fsdisco on a quiescent system.

By experimentation thus far, fsdisco is capable of timing system calls on Unix or Unix-like operating systems. However, a programmer attempting to measure the execution time of individual assembler instructions, would have to resort to platform specific techniques, such as `rdtsc`. Interestingly, this could still be done in a perl script using `Inline::C`; it's just that the resulting script will not be portable to other architectures or perhaps other C compilers.

Admittedly, a perl script's processor and memory [1] requirements are greater than that of a single-purpose executable. To assure measurement correctness, fsdisco carefully avoids the interpreter's overhead in the sections of code that time the interesting system calls.

3 Method & Results

3.1 Platform

The file system and platform we chose to study is ext2 on Linux.[2] We selected this file system partly because (1) ext2 file system documentation is readily available and (2) the debugfs command can be used to examine its structure, enabling us to accurately evaluate fsdisco in preparation for applying it the analysis of less transparent file systems.

The specific system under test (SUT) we chose runs Linux 2.2.18pre21 SMP, has dual Pentium III 450MHz processors, and 128MB of memory. The hard drive involved in the test is an ATA/EIDE Western Digital Corporation WD136AA drive, 13GB in size, with 2MB cache and UDMA. The partitioned drive contains a 249MB swap space and nine ext2 file systems. The results described below were measured on the /tmp file system of 471 MB in size.

The WD Caviar WD136AA hard drive has a read seek time of 9.5 ms typical, 15 ms maximum, and a write seek time of 11.5 ms typical, 17 ms maximum. However, its on-board 2MB buffer has the potential to dramatically reduce read and write measurements.

3.2 Timers

Because the SUT has two processors, timing using the x86 `rdtsc` instruction could be troublesome. This is because the two processors have differing sequences of cycle counter values and it's possible that the measurement process could switch processors during it's execution causing counter discontinuities and adversely affecting our results. Instead, our preference is to measure elapsed time by calculating the interval between calls to the more reliable and portable, but less granular, `gettimeofday()` call.

To test the efficacy of using just the `gettimeofday()` call for fine-grained time measurements, we verified that subtracting the results of a `gettimeofday()` call from a closely subsequent call

[1] fsdisco's resident set size is typically about 3MB due to the perl interpreter itself and a dynamically-sized data structure storing test results.

[2] We also performed some testing of an Apple's HFS+ file system that, like ext2, has a variable block size. This measurement by fsdisco on a PowerPC Mac suggested that its file system's block size was 4096.

can yield time intervals of just single-digit microseconds. As such, it should be possible to differentiate between access to the hard drive, on the order of milliseconds, and access to kernel memory (cache), presumably on the order of microseconds. By experimentation, we further found that it was possible to differentiate between various read and write operations of less than ten of microseconds that didn't involve the disk itself.

3.3 Measuring the File System

The characteristics discovered are summarized in table 1[3].

Characteristic	Value
Block Size	**1KB**, 2KB, or 4KB
Prefetch	approx. 176KB
File Cache	approx. 80MB
Inode Direct Pointers	12

Table 1: ext2 File I/O Characteristics

Tests were performed in single user mode so that the system was quiescent, both with regard to CPU utilization and file cache. Before the test file (/tmp/file.dat) was created, the /tmp file system was reconstructed. The file system was initialized and therefore initially empty, save a lost+found directory. Thus, the test file would occupy a predictable set of blocks and avoid noise in the block access times due to fragmentation.

Also, when run as root, fsdisco remounts the /tmp file system between test iterations in an attempt to flush the file cache. Curiously, this never seemed to cause file content to be flushed from all caches. It is our supposition that the 2MB on-board hard drive buffer was responsible for this effect. To flush all cached content, we found it necessary to reboot the system rather than to just remount the file system. Of course, this was prohibitively cumbersome and slow, so in many cases we used microsecond scale differences in measurement times to deduce behavior because we rarely observed the multi-millisecond times indicative of disk accesses.

3.3.1 Block Size

Our hypothesis was that the ext2 file system blocks size would likely be 4096, as it is a common block size. If not, it would surely be between 512 and 8KB, since those are the smallest and largest block sizes of which we've heard. Research on ext2 informed us that it has variable blocksize, but only three[4] possible block sizes: 1024, 2048, 4096.

[3]We report the prefetch value with some doubt. See section 3.3.2 for details.

[4]Actually, there are four ext2 block sizes: 8192 is used on the DEC Alpha platform.

To discover the block size of our /tmp file system, we ran fsdisco to write (-w) sequentially to /tmp/file.dat at 512 bytes per write, until the file was 1MB in size, and further to do this for 100 iterations (-i 100), unmounting and remounting /tmp between iterations, and writing each iteration's results to a separate output data file (-o write_%04u.log).

```
# fsdisco -w -i 100 -o write_%04u.log
```

Subsequently, we ran fsdisco to summarize (-S) the output data, selecting the minimum (-W min) write time for each 512 byte offset (write position).

```
$ fsdisco -S minimum_write.log -W min write_*.log
```

From these results, we found that the SUT's /tmp ext2 file system initially had a block size of 1024 bytes. This value was automatically selected by mkfs, presumably due to the relatively small size of this file system.

Next, the aforementioned test was performed once for each possible block size on the /tmp file system after it was recreated with that size (mkfs.ext2 -b {1024,2048,4096}). The results are plotted in figure 2.

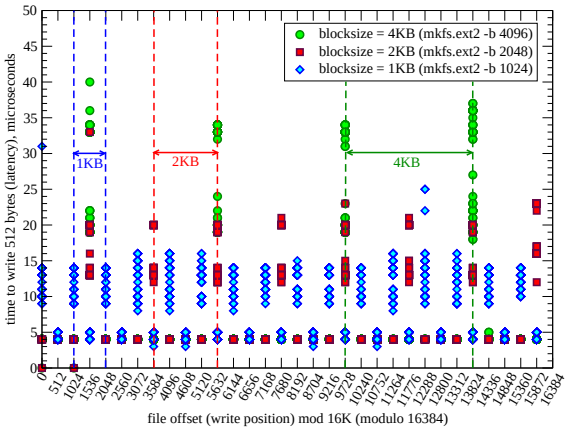

Figure 2: Write latency versus file offset (bytes) mod 16k

Note the vertical clusters of measured latencies. The horizontal distance between them exposes the block size, illustrating that this measurement method correctly identifies all three possible ext2 block sizes.

3.3.2 Prefetch

Our hypothesis was that file I/O on the system under test (SUT) is enhanced to prefetch or "read ahead" some number of blocks in anticipation of continued sequential reads. We suspect this to be observed when a file is opened and

also on subsequent reads. However the situation is complicated; at least two factors could affect prefetch of file data and our ability to reverse-engineer its behavior:

1. Under Linux, the hdparm command can set a drive device-level "read-ahead" feature. On the SUT, read-ahead is enabled for the pertinent drive and is at it's default value of 8 sectors (4KB). However, since the device is not aware of a file's block structure, it will simply read subsequent, contiguous sectors.

2. It is likely that the ext2 implementation itself has some sort of prefetch feature. Although we made a concerted effort to research this on the world wide web, we did not find a concise description of prefetch in the ext2 file system.

Because we've eliminated fragmentation by rebuilding the file system before testing and because ext2 tries to arrange files in contiguous sets of blocks, we expect to observe at least a 4KB "prefetch" effect.

To determine the SUT's prefetch behavior, we first ran fsdisco to create a 3MB file. This file size was chosen because it is larger than the hard drive's buffer, but also small enough not to exhaust the SUT's file cache. The file system's block size was 1024 at the time of this test, we chose to measure appropriately granular 512-byte seeks and reads.

```
$ fsdisco -w -s 3145728 # -s 3MB
```

As noted above, the SUT has a drive with an on-board 2MB buffer and we do not know exactly when and how this buffer is populated. Even if the drive itself doesn't prefetch and just populates its buffer on reads and writes, our analysis should avoid being mislead by this driver buffering. We rebooted the system before the following read test was performed to be sure both Linux' file cache and the drives buffer were initialized to empty.

Then we used fsdisco to read an initial portion of the file (-I 2MB) and then wait five seconds; this is to "prime" the prefetch behavior, in case an initial sequential read is necessary to activate it. This particular value was chosen so that it might fill the drive's buffer as well. fsdisco then read the file *backward* in 1024 byte increments, with the hope of discovering how much of the file past that initial 2MB portion might have been prefetched.

```
# fsdisco -I 2097152 -D 5 -r -b -o backward.log
```

The resulting read latencies are shown in figure 3. Once the read position got within 176KB of the 2MB initial read zone (which is presumably cached), the read times were identical to those below 2MB. This is a strong indication that 176KB of additional file content was cached, possibly due to file system prefetch. This was reproduced in

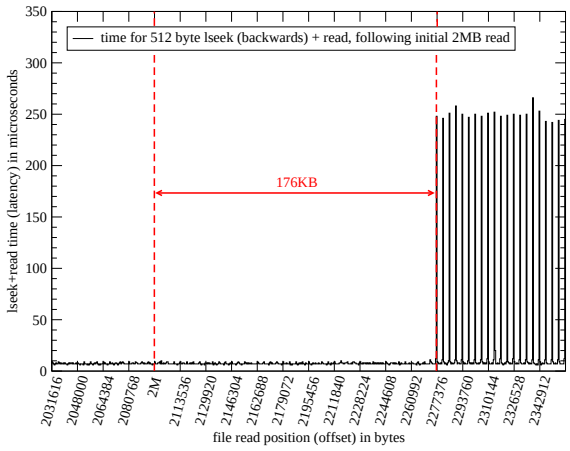

Figure 3: Backward read latency after initial 2MB read

two trials, but we do not have high confidence that this is attributed to, and only to, an ext2 prefetch mechanism.

Also, in figure 3 the reads at higher file offsets, just above that 176KB region, show higher latency at 4KB intervals even though the file system's block size was 1KB. This phenomenon *might* be attributed to the drive device's configured read-ahead size of 4KB or some kernel I/O layer that operates on 4KB objects regardless of block size.

To further explore prefetch behavior, we rebooted (to flush the drive's buffer) and ran fsdisco for 256 iterations with an initial read size of 2MB but had it decrement the initial read size by 4096 on each subsequent iteration. In each iteration we had fsdisco subsequently wait one second (for prefetching) and then read the file backwards as before.

```
# fsdisco -I 2097152 -L 4096 -D 1 -r -b \
-i 256 -o backward_%04u.log
```

We then post-processed the measurement log files to find the extent of the prefetched region as a function of the initial read size. The results of this prefetch analysis are shown in figure 4. We did find evidence of the 176 MB prefetch region seen in figure 3. Unfortunately, we have no satisfying explanation of overall behavior. Our guess is that there is a complicated interaction between the system prefetch and the hard drive buffering that hinders our ability to easily reverse-engineer the SUT's prefetch behavior.

3.3.3 File Cache

Our prior experience was that Linux allows its file cache to occupy all memory that would otherwise be unused. The SUT has 128MB of memory of which the free and top commands report 119MB free immediately after booting

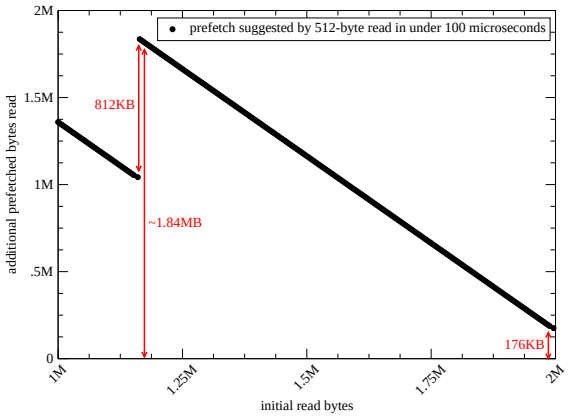

Figure 4: Prefetch extent versus initial "priming" read size

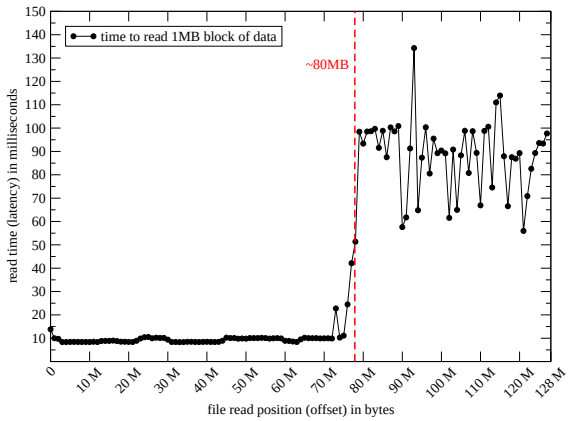

Figure 5: Forward read latency after caching by reading file "backwards"

into single-user mode. Thus, our hypothesis is that the SUT will cache nearly 119MB of a 128MB file when all its blocks are accessed.

Attempting to discover the size of the file cache, we first ran fsdisco to write a file named /tmp/file.dat of the same size as the SUT's physical memory (128MB).

```
$ fsdisco -w -s 134217728 -m 1048576 # -s 128MB
```

After a reboot into single-user mode, we ran fsdisco to read the file *backwards* from end to beginning, as a series of 1MB blocks of data. This action is meant to populate the system's file cache and overwhelm it, leaving the leading part of the file cached since it was most recently read. We ran ten iterations for good measure, in case the system prefers to cache oft-referenced blocks.

```
$ fsdisco -i 10 -r -b -m 1048576 # -m 1MB
```

Subsequently, we ran fsdisco to read the file *forward* from beginning to end, as a series of 1MB blocks of data. We anticipate a jump in latency at the file offset at which it is necessary to access the drive to retrieve uncached file content.

```
$ fsdisco -r -m 1048576 -o forward.log
```

As expected, the file was read much more quickly this time since a portion of the file was cached on prior reads. Figure 5 shows the clear result. The marked increase in read latency at about 80MB into the file strongly suggests that 80MB of this file resided in file cache. Thus 80MB is our rough estimate of file cache memory that was available to cache this file.

3.3.4 Inode Pointers

Having read [1] prior to testing, we had a rough idea that the number of data blocks referenced directly from a file's inode is about ten. So, a test involving a file of less than 100 blocks should be sufficient. During this test the /tmp file system had been created with a 1KB block size.

To discover the number of data blocks that can be referenced directly from the inode, we ran fsdisco to write (-w) whole blocks of data (-m 1024) sequentially to /tmp/file.dat until the file was 64KB in size, and to do this for 100 iterations, unmounting and remounting /tmp between iterations and writing results to separate output data files.

```
# fsdisco -w -s 65536 -m 1024 -i 100 \
-o write_64_blocks_%04u.log
```

Subsequently, we ran fsdisco to summarize (-S) the output data, selecting the median (-W med) time for each block-sized write.

```
$ fsdisco -m 1024 -S median_write.log -W med \
write_64_blocks_*.log
```

The results are plotted in figure 6. Recall that the block size here is 1024 and note the spike in write latency at the 13th data block. This strongly suggests that the first 12 blocks of data are referenced directly in the inode but that writing the 13th block of data required the allocation of a block for indirect pointers first, and then block for the data itself (referenced indirectly) resulting in about twice the write time for that 13th block of data. we used debugfs to verify that it is indeed the case that the 13th allocated block is used for indirect pointers, for a total of 65 1024-byte blocks used to represent the 64KB file:

```
# debugfs -R 'stat file.dat' /dev/hda8
BLOCKS:
(0-11):275-286, (IND):287, (12-63):288-339
TOTAL: 65
```

In figure 6 also note that that the subsequent write times were all shorter, presumably owing to the fact that they

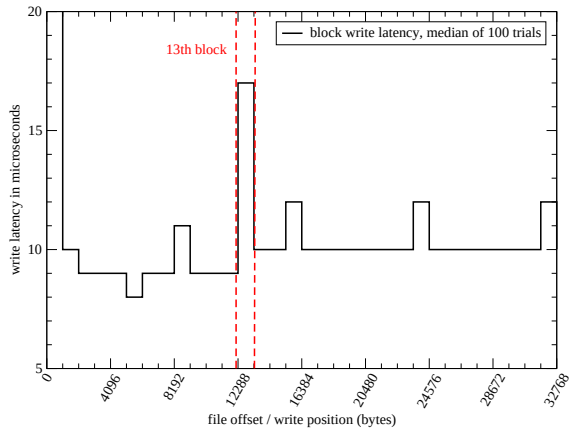

Figure 6: Write latency versus file offset

did not require the allocation of another block for indirect pointers since the one block of indirect pointers was sufficient to represent this 64KB file. This also suggests that, in cache, the indirection itself does not significantly add to the latency, but rather just the allocation and initialization of the indirect pointer block causes a noticeable delay.

4 Future Work

While experimenting with sparse files created by `lseek()`ing past one or more blocks and then writing to subsequent blocks, we discovered that it is slightly faster to read empty blocks. This phenomenon could potentially be exploited to more efficiently divine a file system's block size. For instance, given that ext2 has only four valid block sizes, we believe it would be possible to quickly determine the block size using very few calls to `lseek()`, `write()`, and `read()` by attempting to detect this speed increase when reading across an empty block of one of those candidate sizes.

Also, time and space did not permit a complete report of measurements of the HFS+ file system. That work is a simple matter of using the "recipe" of fsdisco commands in section 3.

5 Conclusion

We have shown that the performance of file systems can be measured to single-digit microseconds using the portable `gettimeofday()` timing facility.

We've learned that it is quite difficult to construct a general tool to perform many specific measurements; fsdisco's myriad command-line options and modes testify to the complication. Furthermore, we were perhaps unnecessarily frugal by performing measurements in a parsimonious manner with respect to the number of system calls per test. Donald Knuth is quoted as having said, "Premature optimization is the root of all programming evil." In this case it was, at least, the root of some unnecessary effort.

fsdisco[5] demonstrates that it is feasible to use a convenient interpreted scripting language such as perl, albeit with extensions written in C, to write and maintain customized tests that effectively measure operating system performance.

6 Acknowledgments

Michael Swift provided the initial motivation for this work and discussed some measurement methods which resulted in fsdisco's test modes and command-line options. Dan Gibson performed a similar investigation of ext2 and his written report[2], that we happened upon whilst searching for background information, was instructive in choosing a test of the inode pointers.

References

[1] R. Card, T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the 1994 Amsterdam Linux Conference*, 1994.

[2] D. Gibson. Measuring Parameters of the EXT2 File System. `http://www.cs.wisc.edu/~gibson/pdf/ext2measure.pdf`.

[3] B. Ingerson. Inline::C. `http://search.cpan.org/`.

[4] C. Walbourn. Game Timing and Multicore Processors. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/Game_Timing_and_Multicore_Processors.asp`, December 2005.

[5] fsdisco is available at: `http://net.doit.wisc.edu/~plonka/fsdisco/`