Measurements of a Unix Filesystem

Block Size, Prefetched Data, File Cache, and iNode Tables

Tyler Simoni
USF College of Engineering
Computer Engineering
Tampa, FL
tylersimoni@mail.usf.edu

Andrew Rodiek
USF College of Engineering
Computer Engineering
Tampa, FL
rodiek@mail.usf.edu

***Abstract*—A Unix filesystem is analyzed to find various attributes of the file system. Using simple system calls in C code - such as read(), write(), open(), and close() to name a few -- the values of some of these attributes can be calculated to relatively good accuracy. In this experiment, the block size, the size of prefetched data, the size of the file cache, and the number of direct pointers in the inodes are found simply by reading or writing to different files and measuring the time of I/O. The motivation of this experiment is to further understand the inner workings of I/O in a Unix operating system.**

I. INTRODUCTION (*HEADING 1*)

In the experiment, an accurate timer first needed to be set up. This was done using the Read Time Stamp Counter. It was used to measure the block size, the amount of prefetched data, the size of the file cache, and the number of direct pointers in the inode.

II. METHODOLOGY

A. Step 1: Timers

In order to accomplish the file system measurements, an accurate time needed to be implemented. In x86 processors starting with the *Pentium*, there exists a 64-bit register called the *Time Stamp Counter (TSC)*.

The timer was implemented via a static inline function called *getticks()*. The function uses inline assembly language to first set the cpuid. The CPUID call allows the processor to wait for all other instructions to be read before reading the counter, as this relies on the CPU clock cycles. The RDTSC call counts the number of clock cycles since reset and "clears the higher 32 bits of RAX and RDX" (Intel® 64 and IA-32 Architectures Software Developer's Manual). Once this was done, it was converted to milliseconds by dividing by the clock frequency over one thousand.

This timer was confirmed accurate by using the reliable, yet less accurate timer, *gettimeofday()*.

B. Step 2: Measuring the file system

1) Block Size

The first measurement taken was for the block size of the disk. This was done by creating files of various sizes and testing the time to read each of them. The idea was that after the block size was hit, there would be a spike in time for each subsequent file size read until the next full block was hit.

The files were created in the C code by using *fallocate()* to create files full of null bytes of sizes 1, 2, 4, 5, 6, 8, and 16 kilobytes. Each of these files were repeatedly created, opened, read, and closed, and removed. The average of 256 times of the read cycles were taken and used for the data results. The amount of loop iterations (256) was chosen arbitrarily.

2) Amount of Prefetched Data

The second measurement was done to find the amount of data that was prefetched by the file system. The idea for this part of the experiment was that the file system would take much longer to open the file initially and then each subsequent read would be much less and equalize out. This would be due to the file system prefetching data so that it does not have to perform a raw I/O from the disk each time. This was done by performing a sequential read of a 1024 megabyte (~1 gigabyte) file created using *fallocate()*.

The C code created the file, then in a loop, opened, read, and closed the file, 256 times. The amount of loop iterations (256) was chosen arbitrarily.

3) Size of File Cache

The third measurement was done to deduce the size of the file cache of the file system. The idea for this part was to look for the spike in read times between the different amounts of data in a large file. Once a significant spike in time was found, it was deduced that read had exceeded the file cache size and had to go back to disk to read in more data.

This was completed by reading in a 2 gigabyte file – again created using *fallocate()* – and reading different amounts of its data at a time, starting at 32 kilobytes and doubling the size of the file up to 32 megabytes.

4) Number of Direct Pointers in the iNode

The fourth and final measurement performed in this experiment was to approximate the number of direct pointers in each inode of a file. The idea behind this measurement was to create an empty file and append one block at a time to the end of the file.

The time it took to append each block was measured and when an increase in the time to write to the file was seen, it could be deduced that the number of direct pointers had been

be crossed. This is because a write to an indirect pointer would take more time.

III. Results

A. Block Size

The read times from the block size test yielded successful results. After many trials of running the test program on the file system, an initial spike in time was noticed between 4 and 5 kilobytes. After this, the time grew fairly proportional to the size of the file being read in.

From this, we deduced that the block size was around 4 kilobytes, or 4096 bytes. We used the *stat* command to find out that the block size of the machine was in fact 4096 KB. *Below* the data and a graph showing the time increase.

Size(KB)	Time(ms)
1	3502
2	4239
4	4512
5	5759
6	5805
8	6255
16	9060

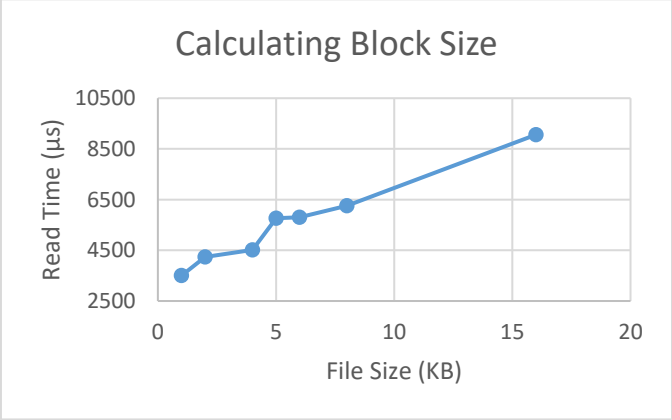


Figure III-1. Block Size Test Data

B. Prefetch Data

After repeatedly reading the same file without removing it, a noticeably shorter time to read was seen after the first read. Because the file had already been fetched, sequential reads progressively became shorter.

Starting at an initial read time of about 620ms, the next read dropped to just 94ms. After that it got even shorter. This would be in part due to the repeated reading of the file. The file system was predicting that the same file would be opened again and so would fetch more data. After only 19 read iterations, it leveled out at around 69ms. From this time data, we calculated that about 80% of the data was being fetched, by dividing the shorter time by the initial read time.

However, because the file being read was 1GB, this would mean the file system was prefetching almost 800MB of data for this file. Unless the file system prefetched dynamically to a huge degree, this seemed improbable. Thus, based on the results, this part of the experiment was deemed unsuccessful. The data is shown below:

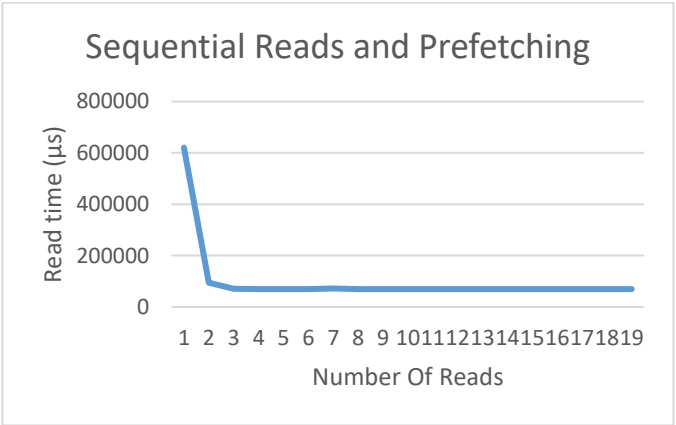


Figure III-2. Prefetch Test Data

C. Cache Size

The largest initial spike in read times were from 512 kilobytes and 1 megabyte. From this we assumed that the cache size was about 512 kilobytes. This would then explain the subsequent jumps in time between the different read sizes of the 2 gigabyte file.

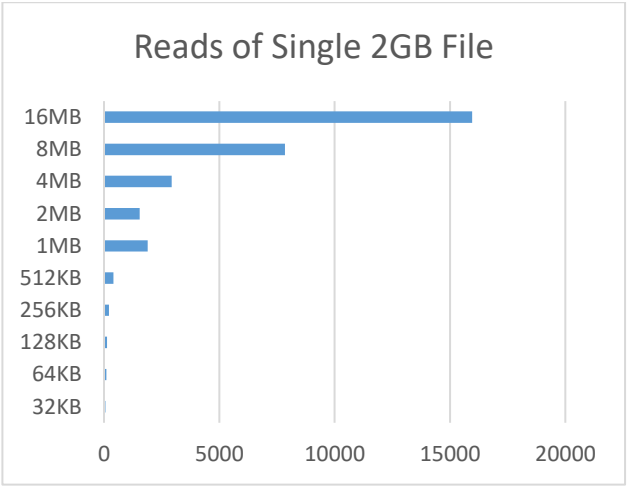
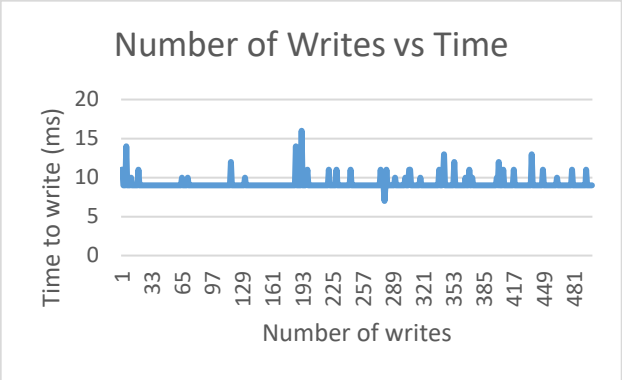


Figure III-3. Cache Size Test Data

The problem with this data is that the *lscpu* command in the bash shell on the C4 Linux machines shows that multiple cache sizes exist. They are as follows; L1d cache: 32K; L1i cache: 32K; L2 cache: 256K; L3 cache: 8192K. Also reading the */proc/cpuinfo* file using *cat* showed a single cache size of 8192 KB. None of these sizes matched our hypothesized cache size and so the cache size test was unsuccessful.

D. *Number of Direct Pointers in iNode*

For this experiment, the text file was created and written to repeatedly using the *write()* function call. After each write, *fsync()* was used to push the data out of the buffer and onto the disk. We kept of the number writes versus the time.



Once this data was obtained, the spread seemed fairly random, as the time between spikes was not very uniform.

The idea we had in mind was to write to the disk repeatedly, one block at a time. When a spike in time was seen, one could assume that we had exceeded the number of direct pointers due to the extra time of I/O from seeking the indirect pointer to a block.

Due to this, this experiment yielded unsuccessful. We had no further ideas on how to obtain the number of direct pointers to blocks in the inodes.

IV. Conclusions

We spent a lot time trying different methods on how to get the information. Unfortunately, most tests yielded unsuccessful. With a more in depth knowledge of the inner workings of the file system, better experiments could be performed that would produce more accurate and possibly precise data.

V. Time Spent

Approximately 37 hours

VI. Team Member Participation

A. *Tyler*

- Coded the skeleton of the program including the main, function setups, and initial timers.

B. *Andrew*

- Coded the block size, prefetch, cache size, and inode functions

C. *Full collaboration*

- Report in IEEE format
- Research on file system information