File System Measurements

Kiranmayi Karri
Dept. of Computer Science
University of South Florida
Tampa, Florida, USA
kiranmayi1@mail.usf.edu

Sai Kiran Reddy Malikireddy
Dept. of Computer Science
University Of South Florida
Tampa, Florida, USA
malikireddy@mail.usf.edu

ABSTRACT

File System is a service which supports an abstract representation of the secondary storage to the OS. This provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily. Performance of file system can be determined by measuring their factors . These measurements help us to understand the behavior of the file system. In this project, we successfully determined certain measurements of file system such as block size, prefetch size, cache size and how many direct pointers exists. Also, the results from our experiments were analyzed and reported accordingly.

1 INTRODUCTION

Accessing data from disk is an essential feature of any operating system. The data on the disks are stored in a structure of files which allow the data to be accessed efficiently. File system is the way which provides these files to be arranged on disk in a hierarchy. Measuring the parameters related to file system allows one to evaluate the performance of the system. In this report, we provide with the approach of how we analyzed and measured certain parameters of the file system. We have experimented our measurements to address the following requirements.

- Block Size used by the file system.
- Prefetching size of the file system.
- File Cache Size.
- Number of direct pointers in the inode.

To achieve these requirements, we have followed few methodologies which have been discussed in further sections. The results were explained and reported for each of the above question posed. Our report is organized as follows, the next section gives, the approaches/ methodologies which we followed in order to successfully achieve the project requirement. Results section which gives detailed evaluation of our experimental results. Followed by conclusion and references.

2 METHODOLOGY

In order to achieve accuracy of the file system measurements, we have taken several things into consideration such as the platform, timers, type of file systems, system calls.

- **Platform :** Our measurements were experimented on LINUX system with Ext2, Ext4 filesystems with operating system as Ubuntu.
- **Timers :** To ensure accuracy and granularity of our system measurements, timers play a critical role. we have used highly accurate cycle counter called rdtsc() which is provided by x86 platform. rdtsc() is a time stamp counter which returns the number of CPU cycles. We have then converted the clock cycles to microseconds in the code. Also, we have measured the parameters using gettimeofday() timer and compared the results to increase our confidence.
- **Conversion formula for rdtsc() :**

Time in microseconds = (1/processor speed(Megahertz))*number cycles

- **File System :** To abide to the project requirements of not using the distributed file system, we haven't performed our experiments on the C4 lab machine, instead we used our personal system with LINUX operating system. There were certain scenarios where we had to use ext2 and ext4 file systems for taking measurements. Further details are given in the following sections.
- **System Calls** : We used some system calls like read(), lseek(), write(), fsync() in our experiments.

Before going into further details of the measurements, we discuss the configurations of the system which was used to perform the experiments.

2.1 System Configuration

2.1.1 Hardware Environment

Configurations			
Processor	Intel Core I5		
Processor Speed	2.50Ghz		
No. of Cores	2		
Memory Size	8 GB		
Linux	Ubuntu 14.04		
Kernel Version	3.16		
Caches	L1	L2	L3
	32KB	256KB	33MB
Hard Disk	750GB with 5200 RPM		

2.1.2 Software Environment

Operating System : Linux

Compiler : gcc

Language : C programming

File System **:** Ext2/ Ext4

2.2 Measurements

2.2.1 Timer Accuracy

In this project, timer accuracy plays a key role while taking the measurements. To find the accuracy, we have tested two different timers (rdtsc(), gettimeofday()), by issuing sleep for 10 sec in between the start and end of timers. From our experiments, we found that rdtsc() is the most accurate timer. Results can be referred at Timer_Results.

2.2.2 Block Size Measurement

Block size can be defined as the size which file system utilizes to perform read or write operations of the data. The main advantage of larger block size is, it will enhance the performance of the disk I/O when there is a usage of larger files. Though , larger block size improve the system performance, we also have a disadvantage of it where when we have huge number of smaller files in the file system, it would waste the disk space. Nowadays, we have operating which are restricted by number of blocks but not block size.

- **Approach**

The proposed is the solution which we used to measure the block size of the file system. We start accessing the file in sequential order. Here we used lseek() system call to set the file descriptor to start of the file. This system call sets the position of file descriptor to the given offset. We measure the time taken to perform this operation by using rdtsc() before and after the read call. This approach is repeated for 100 times and averaged the result to increase the accuracy in our results.

- **How the size is determined :** The block size has been decided based upon the time taken for accessing the blocks. When there is a context switch operation for varying sizes of reads i.e., switching from one block to another, there was a steep increase in the time taken to access the next block.

- **Challenges**

How to avoid prefetch or readahead mechanism in sequential read?

In order to avoid such activity by the operating system, we have implemented the usage of **posix_fadvise().** This access the file in certain pattern, and thereby allowing the kernel to perform optimizations.

```
posix_fadvise(fd,0,0,POSIX_FADV_RANDOM);
```

POSIX_FADV_RANDOM disables prefetch or readahead mechanism entirely. This change will be effected through entire file and not just that specific region[4].

How to prevent caching the data in the cache buffer for calculation of accurate size?

Caching is the behavior where the frequently accessed data is stored in the cache for the effective access. In order to avoid caching we have implemented the usage of **posix_fadvise().**

```
posix_fadvise(fd,0,0,POSIX_FADV_DONTNEED);
```

POSIX_FADV_DONTNEED free the cached pages which are associated with that region[4].

Expected results: We know that the valid block sizes for a ext2 file system are 1K,2K,4K [2]. So, we expect the result to be either of one.

Actual results: Can be referred at block_size_results .

2.2.3 Prefetching Size of File System

Prefetch: Whenever the requested data is not found in the cache, then the operating system issues the request to disk for appropriate block[6]. If the system identifies the access to be sequential then it prefetches the additional data. The amount of data prefetched is doubled on each disk read, up to a maximum of the cluster size. The main advantage of this mechanism is it improves the memory latency.

- **Approach**

The proposed is the solution for calculating the prefetch size of the file system . To boost the prefetching mechanism in the file system, we initially accessed the file for certain number of bytes using read() system call. And then we used sleep() system call for 1sec to allow the system to complete prefetching operation of the data. Now, access the file sequentially by reading the data in units of certain number of bytes.

- **How the prefetching size is determined:** This can be concluded by observing the time measurements while accessing the file. If the time taken to read certain number of bytes is low, then we can say that the data is already prefetched, else it takes more time to read the data from the disk. The prefetch size will be the number of bytes between the peek time of accessing.

- **Challenges**

The main challenge which we faced while determining this size was to avoid caching of the data. As said above, we would conclude the size based upon the time taken to access the data read. If the data is already cached then it will be take less time to access the data which may mislead our calculation. Hence we need to avoid caching of the data.

- **How to avoid caching the data in the cache buffer for calculation of accurate size?**

To avoid caching we have used **posix_fadvise().**

```
posix_fadvise(fd,0,0,POSIX_FADV_DONTNEED);
```

POSIX_FADV_DONTNEED free the cached pages which are associated with that region[4]

Actual results: Can be referred at prefetch_size_results .

2.3.4 File Cache Size of File System

Cache is the storage area where the recently accessed data will be stored, in few cases it might also store the adjacent data which is likely to be accessed next. There are some caches which provide the mechanism of write caching. The main advantage of this mechanism is a cache reduces access time of the data. If there is a

cache hit then we can say that the access time of that data is equal to the access time of the cache.

Two types of cache are memory cache and disk cache. Memory cache is a portion on memory of high-speed static RAM and is effective because most programs access the same data or instructions over and over. Like memory caching, disk caching is used to access commonly accessed data. However, instead of using high-speed SRAM, a disk cache uses conventional main memory[11]. The most recently accessed data from a disk is stored in a memory buffer. When a program needs to access data from the disk, it first checks the disk cache to see if the data is there. Disk caching can dramatically improve the performance of applications because accessing a byte of data in RAM can be thousands of times faster than accessing a byte on a hard drive.

- **Approach**

The proposed is the solution for calculating the cache size of the file system . First, we read file for certain number of bytes. As these bytes are recently accessed, they will be stored in the cache. In the next step we try to access the file in the reverse order for each 4KB and measure the time taken.

- **How the cache size is determined:** In this process, when the data is read for the first time the data will be cached, here if the cache size is less than the size of the data accessed then certain data will be replaced with the newly accessed data. Now when we start accessing the data in the reverse order then the data , if we have a cache hit then the time taken to access will be less than the time on cache miss. In this way wherever the time increases, we can determine the cache size.

- **Challenges**

There were many challenges faced during the measurements taken for file cache size. Due to these issues, we have observed some inconsistency in our results. The inaccuracy was due to many external factors.
The first challenge here is clearing the caches, this was implemented using many linux commands.

- *sudo blockdev --flushbufs /dev/sda1 , this command will flush the disk cache.*
- *sudo blockdev --setra 0 /dev/sda1, this sets the readahead or prefetch size to 0 which means that we are disabling the prefetch mechanism.*
- *sudo sh -c 'echo 3> /proc/sys/vm/drop_caches' , used to clear the data in these caches.*

Apart, from the execution of the above commands, we have filled the RAM space with 90% of the data, so that we assure that the data is not accessed from RAM.

The other challenge is that if the file size is less than the cache size, we cannot estimate the cache size as there will be only cache hits and no misses which would take almost equal time to access the complete data.

How does one ensure that file size is greater than the cache size?

We have tested the code with various files ranging from 100MB to 800MB by incrementing the file size for every 100MB.

Note: Also, ensure that the cache is not flushed between the successive reads as this would result in cache miss and time will be huge for each access.

Expected results: Using linux command , we found that cache size is 16MBytes.

Actual results: Can be referred at **cache_size_results** .

2.3.5 Number of Direct Pointers

Inode Pointers : Every entry in a file system contains the pointer to the respective file's inode. The metadata of this file is stored in the corresponding inode. The first few blocks which is being pointed by the inode itself is called direct blocks. If the inode points to an indirect block, this in turn then points to disk blocks. The inode also points to a double indirect block, which points an indirect block, which points to data blocks[15]. For some implementations there is a triple indirect block as well.

- **Approach**

The proposed is the solution for calculate the number of direct pointer in a file system. In this process, we create a new file and start writing the number of bytes equal to block size in the file. Here the block size can be determined from the first experiment. Whenever the size of file exceeds the limit of direct blocks, then the time increases as the system has to write the data now into indirect block. This means it takes time which is equal to two writes.

- **Challenges**

Here the main challenge was when tested on ext4 file system , there was no consistency in the result. This was due to the concept that ext4 doesn't use any blocks such as direct or indirect. There is a new concept of 'extents ' being introduced.

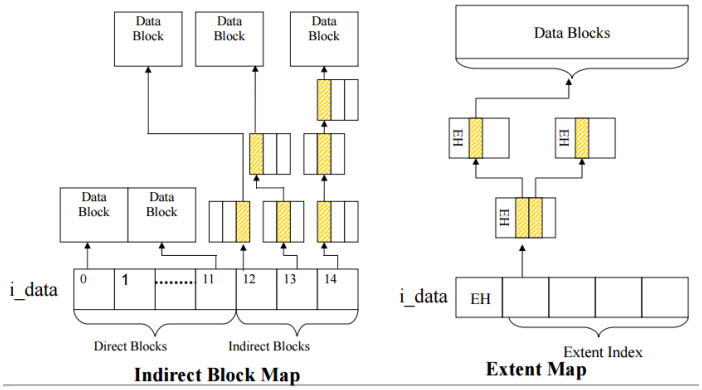

Fig 2.1 Comparison between Ext2 and Ext4 [18]

Hence, to find the direct block count, we mounted to ext2 file system and taken the measurements.

Expected results: As we are aware that in ext2 file system , it has 12 direct block pointer, we expect the actual result to be same.

Actual results: Can be referred at direct_blocks_results .

3 EXPERIMENTAL RESULTS

This section provides the results, for the experiments described in the section 3. As mentioned, we have taken certain precautions to determine accurate results as well as ensuring that we met the given project requirements.

3.1.1 TIMER RESULTS

Program name: timer.c, **Output File:** timer.txt

Code Snippet

```
1    float get_cpu_clock_speed ()
2    {
3    FILE* fp;
4    char buffer[1024000];
5    size_t bytes_read;
6    char* match;
7    float clock_speed;
8    system("cat /proc/cpuinfo > out.txt");
9    fp=fopen("out.txt","r");
10   bytes_read = fread (buffer, 1, sizeof (buffer), fp);
11   fclose (fp);
12   if (bytes_read == 0 || bytes_read == sizeof (buffer))
13   return 0;
14   buffer[bytes_read] = '\0';
15   match = strstr (buffer, "cpu MHz");
16   if (match == NULL)
17   return 0;
18   sscanf (match, "cpu MHz : %f", &clock_speed);
19   return clock_speed;
20   }
```

Fig 3.1 Code snippet for retrieving processor speed of the system

```
int i=0;
struct timeval start,end;
/******get the processor speed***************/
        float mhz=(get_cpu_clock_speed());
/*********calculate time for rdtsc**********/
        unsigned long long before = rdtsc();
        sleep(1);
        unsigned long long after  = rdtsc();
/*********convert cycles to microseconds******/
        double final=(after-before)/(mhz);
        printf("time of rdtsc %lf\n",final);

/*********calculate time for gettimeofday*****/

        gettimeofday(&start, NULL);
        sleep(1);
        gettimeofday(&end, NULL);
        printf("%ld\n", ((end.tv_sec * 1000000 + end.tv_usec)
          - (start.tv_sec * 1000000 + start.tv_usec)));
        return 0;
        }
```

Fig 3.2 Code snippet for comparing the timers

This code retrieves the CPU information using the command '/proc/cpuinfo' and stores the information in a text file. Then we read this file for processor speed with the keyword 'cpu Mhz' and then parse the line to retrieve the numerical value of the speed and store it in a variable clock_speed. This value is used to convert the rdtsc cycles to microseconds.

Result

We observed that rdtsc is more accurate than the gettimeofday() time, hence we have used rdtsc() for calculating are measurements.

Graph

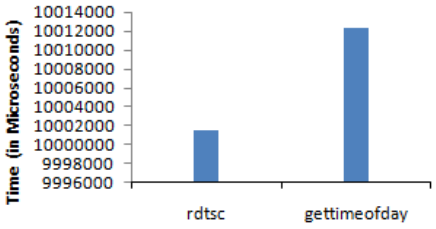

Fig 3.3 Graph comparing the timers

3.1.2. BLOCK_SIZE_RESULTS

Program name: block_size.c,
Output File: blocksize_average.txt

Code Snippet

```
1    posix_fadvise(fd,0,0,POSIX_FADV_DONTNEED);
2    posix_fadvise(fd,0,0,POSIX_FADV_RANDOM);
3    lseek(fd,0,SEEK_SET);
4    for(int j = 0; j < 50 * KB ; j = j+512 )
5    {
6    unsigned long long before = rdtsc();
7    read(fd,buffer,min_blocksize);
8    unsigned long long after = rdtsc();
9    if(j != 0)
10   {
11       double elapsed=(after-before)/mhz;
12       sprintf(output,"%d %f" , j , elapsed);
13       outputarr[i]=puts(output);
14       i++;
15   }
16   }
17   close(fd);
18   return 0;
19   }
```

Fig 3.4 Code snippet for block size measurement

The above is the code snippet for measuring the block size(we numbered the code for our convenience to brief it). Lines 1,2 refer to the predefined function which we used to clear the disk cache and disable the prefetch mechanism. In line 3, we used lseek to set the file position .From line 4 until end we repeated the process of reading the file sequential order in units of 512 bytes and measuring the time taken for this process. Lines 11 converts the elapsed clock cycles to microseconds and the next lines prints the output.

Graph

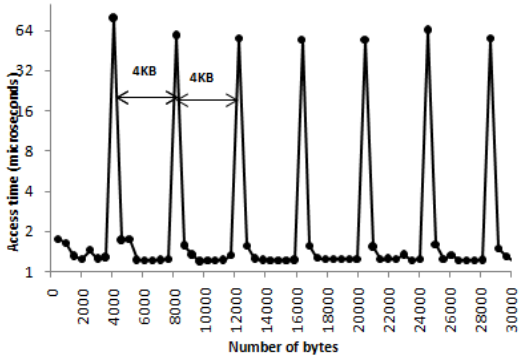

Fig 3.5 Measurement of block size

Result

From, the above graph we can conclude that for every interval of 4096, there is a steep increase in the access time. Hence, we say that the block size of file system is **4096Bytes or 4KB.**

Precautions

Caches are cleared before every run. Avoided random read of file.

Accuracy

To improve accuracy of our result, we have repeated this experiment for 100 times and took an average of it. Here the complete program is repeated using a for loop specified in make file.

```
blocksize: block_size.exe
	@for (( i = 0 ; i < 100; i = i + 1 )) do ./block_size.exe $(FILELOC);
```

Verification

We have cross verified our result on the tested system through a linux command and confirmed our result.

```
kiranmayi9@ubuntu:~/Desktop/pro4$ sudo blockdev --getbsz /dev/sda1
4096
```

3.1.3 PREFETCH_SIZE_RESULTS

Program name: prefetch_size.c,
Output file : prefetchsize_file1MB.txt
Code Snippet

```
1	posix_fadvise(fd,0,0,POSIX_FADV_DONTNEED);
2	unsigned char *buffer=malloc(sizeof(*buffer)*buffer_size);
3	while(bytecount < k * MB)
4	{
5		read(fd,buffer,buffer_size);
6		bytecount += buffer_size;
7	}
8	sleep(1);
9	bytecount = 0;
10	buffer_size = 10 * KB;
11	free(buffer);
12	buffer[buffer_size];
13	printf("Kilobytes" "\t" "Time(microsecs)""\n");
14	printf("====================================""\n");
15	while(bytecount < MB)
16	{
17		unsigned long long before = rdtsc();
18		int dataread = read(fd,buffer,buffer_size);
19		unsigned long long after  = rdtsc();
20		double elapsed=(after-before)/mhz;   //converting cycles to microseconds
21		sprintf(output,"%d \t\t %f" , bytecount/KB,elapsed );
22		outputarr[i]=puts(output);
23		i++;
24		bytecount += dataread;
25	}
26	close(fd);
27	return 0;
28	}
```

Fig 3.6 Code snippet for prefetch size

Lines 1 refer to the predefined function which we used to clear the disk cache. Line 3 to 6 will read the given file in units of 1KB each time. Here, we issued sleep call for 1 second in order to allow the system to complete the prefetch mechanism. Now read the data from the file for 10KB each time. Line 20 converts the elapsed clock cycles to microseconds and the next lines prints the output.

Results

From the below graph, we can see that the first peak of access time occurs at 112KB .Considering the average of all the intervals i.e., 127KB,124KB,127 KB etc., we could say that the prefetch size is equivalent to **128KB.**

Graph

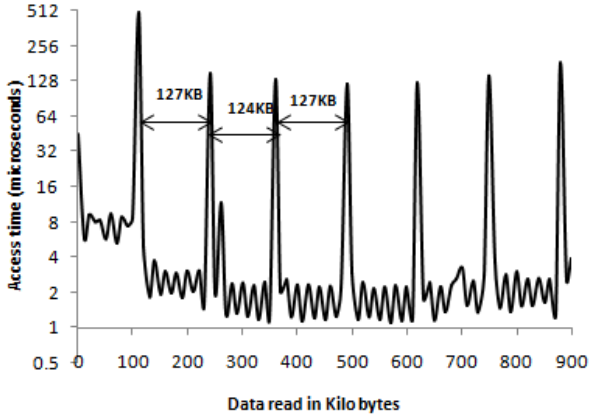

Fig 3.7 Measurement of prefetch size

Precautions

Cleared the cache before each run of the program.

Accuracy

To be confident of our measurement , we have executed the program with varying sizes of the input files such as 1MB,2 MB,4MB. It was seen that the result is close to **128KB.**

Verification

Cross verified the result, using the below linux commands.

```
kiranmayi9@ubuntu:~/Desktop/pro4$ sudo blockdev --getra /dev/sda1
256
```

The above command gives the result in **sectors**.

```
kiranmayi9@ubuntu:~/Desktop/pro4$ sudo fdisk -l

Disk /dev/sda: 21.5 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders, total 41943040 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
```

From the above screen print , we can see that the sector size is 512 bytes in the system , hence the prefetch size with **256 sectors** will be **131072 Bytes** which equivalent to **128KB.**

3.1.4 FILE CACHE_SIZE_RESULTS

Program name: filecache_size.c**, Output file :** filecache_size.txt

In the code snippet, Line 1 thru 6 reads the complete file in sequential order. Once the file has been read without any errors. We read the file in the reverse order to find the cache size. Our basic idea here is, whenever we perform the first read of the file, if the cache size is less than the file size, initially read blocks of data will be replaced by the recently read pages. Hence when we access the file for the second time in the reverse order, there might be a cache miss which can be determined if there is a sudden increase in the access time. As, if there is a cache miss the block of data will be accessed from the disk which takes more time.

Code Snippet

```
1       while(true)
2       {
3       int dataread = read(fd,buffer,buffer_size);
4       if(dataread <= 0)
5               break;
6               size += dataread;
7       }
8       printf("Block no.""\t""Time(microsecs)""\n");
9       printf("========================================""\n");
10      while(size > 4 * KB){
11      lseek(fd,-buffer_size,SEEK_CUR);
12      unsigned long long before = rdtsc();
13      read(fd,buffer,buffer_size);
14      unsigned long long after= rdtsc();
15      double elapsed=(after-before)/mhz;
16      read_blocks = read_blocks + 1;
17      sprintf(output,"%d \t\t %f" ,read_blocks,elapsed );
18      outputarr[i]=puts(output);
19      i++;
20      lseek(fd,-buffer_size,SEEK_CUR);
21      size -= buffer_size;
22      }
23      posix_fadvise(fd,0,0,POSIX_FADV_DONTNEED);
24      close(fd);
25      return 0;
26      }
```

Fig 3.8 Code snippet for file cache size

Graph

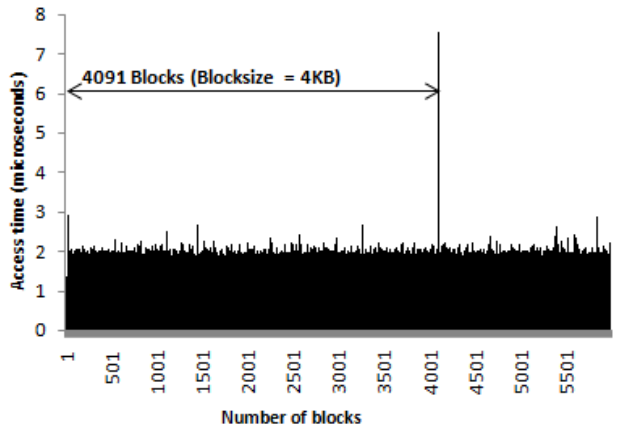

Fig 3.9 Measurement of file cache size

There was a peak increase in the time when the system accessed 4091 block.

We know that each block size = 4 KB, hence 4091 blocks = 4091*4KB = 16364 KB which equivalent to 16MB.

Result

There was a steep increase in the access time after an approximate access of **15.98 MB** data. But there were many external factors which were described in the previous section (refer to section 2) which impacted this measurement.

Accuracy

As we have no idea about the cache size, we experimented our implementation with various sizes of files from 100MB to 1GB.

- **Precautions**

As described previously, while measuring the cache size we have many factors which impacts the result. In order to get an accurate result, we have disabled the prefetch functions, cleared all the caches before experimenting each run using many methods. One was to unmount and mount the file system for every run of this program so that the caches will be completely cleared, apart from this process we also used below linux commands to clear the caches. Used hardware tools such as msrtools to disable. The other factor here is storage in RAM, hence we filled 90% of the RAM using linux commands.

Mount and Unmount

```
kiran@kiran-HP-ProBook-450-G1:~$ sudo mount /dev/sda5 ./Desktop/pro
kiran@kiran-HP-ProBook-450-G1:~$ df -T
Filesystem     Type     1K-blocks    Used Available Use% Mounted on
/dev/sda6      ext4     148141496 23035224 117558012  17% /
none           tmpfs            4        0         4   0% /sys/fs/cgroup
udev           devtmpfs   4016748        4   4016744   1% /dev
tmpfs          tmpfs       805508     1308    804200   1% /run
none           tmpfs         5120        0      5120   0% /run/lock
none           tmpfs      4027524  4027524         0 100% /run/shm
none           tmpfs       102400       56    102344   1% /run/user
/dev/sda5      ext2     100790704 11244608  84426148  12% /home/kiran/Desktop/pro
```

To fill the RAM upto 90% , we used the below linux command[14]

```
swapoff -a
dd if=/dev/zero of=/dev/shm/fill bs=8k count=1024k
```

Verification

Once, we have experimented with this method , we verified our result by checking the file cache size of the tested system using linux command.

sudo hdparm -I /dev/sda5

```
Logical Sector-0 offset:                          0 bytes
device size with M = 1024*1024:      715404 MBytes
device size with M = 1000*1000:      750156 MBytes (750 GB)
cache/buffer size  = 16384 KBytes
Nominal Media Rotation Rate: 5400
```

3.1.5 DIRECT_POINTER_RESULTS

Code Details: direct_pointer.c**, output file :** direct_pointer_average.txt

Code Snippet

```
1       while(size < 100 * KB)
2       {
3               unsigned long long before = rdtsc();
4               write(fd,buffer,4 * KB);
5               fsync(fd);
6               unsigned long long after = rdtsc();
7               double elapsed=(after-before)/mhz;
8               blocks_read=blocks_read+1;
9               sprintf(output,"%d %f" , blocks_read , elapsed);
10              outputarr[i]=puts(output);
11              i++;
12              size += 4 * KB;
13      }
14      close(fd);
15      return 0;
16      }
```

Our idea here was to create a file (line 1 thru line5), then perform write operation with 4KB(block) each time(line 7 to line9) and measure the time taken to perform this using rdtsc(). Here we

used fsync() system call, this transfers data from buffers to the file to the disk device using the file descriptor[14]. Whenever the size of written data is greater than the size of the direct blocks, it jumps to indirect pointer blocks. This increases the time to write. This was tested on Ext2 file system as the direct pointer block concept is not available on Ext4.

Graph

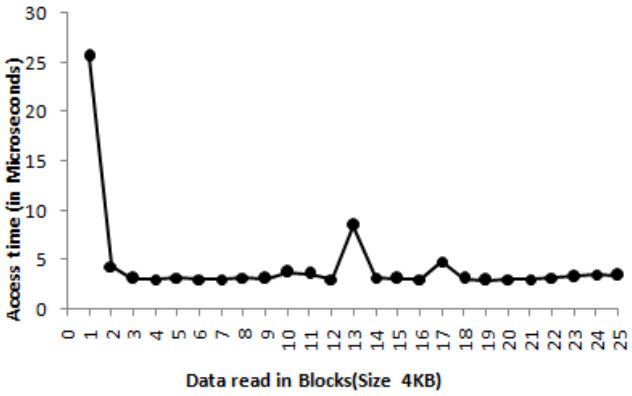

Fig 3.10 Measurement of Direct pointer blocks size

Result

From the above graph, we observed that there is a steep increase at the time of 13th block access, hence we can say that the mechanism triggered from direct block to indirect block. This resulted in the sudden increase in the time. So, we can conclude that there were **12 direct pointers.** Whenever there is switching performed from direct to indirect block, it is equivalent two writes.

Accuracy

To improve accuracy of our result, we have repeated this experiment for 10 times and took an average of it. Here the complete program is repeated using a for loop specified in make file.

```
directpointer: direct_pointer.exe
	rm -Rf $(FILELOC)/direct_pointer_*.txt
	@for (( i = 0 ; i < 10; i = i + 1 )) do ./direct_pointer.exe $(FILELOC)
done
```

Precautions

As mentioned in section 3, there is no concept of direct pointers/indirect pointers in ext4 file system. So, we have mounted our file system to ext2 and then executed the code on this type of system.

Verification

We know that an ext2 file system has 12 direct pointer, 3 indirect pointer[15].

4 CONCLUSION

In this paper, we reported measurements that we experimented on the ext2/ext4 file system and analyzed the results. We found the results are nearly equivalent to the expected results which we found using linux commands. The block size of the file system is determined to be 4KB. Prefetch size is nearly equivalent to 128KB. File cache size has been found as 16MB and the number of direct blocks were 12 on Ext2 file system and there is no concept of direct pointer blocks in Ext4 file system.

5 REFERENCES

[1] http://www.mcs.anl.gov/~kazutomo/rdtsc.h

[2] http://www.linfo.org/get_block_size.html

[3] http://linux.die.net/man/8/mkfs.ext3

[4] http://linux.die.net/man/2/posix_fadvise

[5] http://unix.stackexchange.com/questions/43279/disabling-readahead-with-hdparm-or-posix-fadv-random

[6] https://www.usenix.org/legacy/event/usenix99/full_papers/shriver/shriver.pdf

[7] http://pages.cs.wisc.edu/~madhurm/ext4FileSystem

[8] http://linux.die.net/man/2/read

[9] http://e2fsprogs.sourceforge.net/ext2intro.html

[10] http://pages.cs.wisc.edu/~gibson/pdf/ext2measure.pdf

[11] http://www.computerhope.com/jargon/c/cache.htm

[12] http://www.tldp.org/LDP/sag/html/filesystems.html

[13] http://linux.die.net/man/2/gettimeofday

[14] http://unix.stackexchange.com/questions/99334/how-to-fill-90-of-the-free-memoryhttp://man7.org/linux/man-pages/man2/fsync.2.html

[15] http://people.cs.umass.edu/~trekp/csc262/lectures/04.pdf

[16] http://unix.stackexchange.com/questions/49624/average-rows-with-same-first-column

[17] http://www.thegeekstuff.com/2011/05/ext2-ext3-ext4/

[18] https://events.linuxfoundation.org/slides/2010/linuxcon_japan/linuxcon_jp2010_fujita.pdf