

# Exploring and Measuring Unix-Based File Systems

Liliana Aguirre Esparza  
NMSU, CS Department  
Las Cruces, NM  
lili91@nmsu.edu

Jose Franco Baquera  
NMSU, CS Department  
Las Cruces, NM  
jose5913@nmsu.edu

**Abstract** – Exploring the inner-workings of a Unix-based file system can be done by writing small C programs that exercise the file system in question in different ways. These small programs can use system calls, such as `read`, `fsync`, `write`, `open`, and `close`, to determine how long certain file system operations might take, and allow us to deduce what actions are being performed by it. This paper summarizes four experiments that were run to determine certain characteristics of *hopper*, a New Mexico State University (NMSU) host. It was found that *hopper*'s file system uses a block size of 4096 bytes, and that 14 blocks are prefetched into cache by its file system. It was also determined that the cache size is approximately 14GiB, and that the file system uses an extent-based allocation method with an extent size of approximately between 83 and 84MiB.

**Keywords** – file systems; block size; prefetched data; main memory cache size; read; write; `fsync`; allocation method; extent size

## I. INTRODUCTION

An operating system contains many system programs, data structures, daemons, and other tools that allow it to perform its functions in an effective way. One of the most important tools utilized by operating systems is the *file system*. For many users, the file system is the most visible feature of the operating system since it is the structure with which they interact the most. A file system is a set of methods and data structures that provides a mechanism for online storage of and access to the programs and relevant data used by both the OS and system users. The file system usually consists of two separate parts: a directory structure used to organize and provide file information, and a set of files, each of which stores relevant data related to the file. The file system permanently lives in non-volatile secondary storage devices such as hard disks. Therefore, when a file is requested by a user, it must be retrieved from the disk and placed in main memory, where it can be accessed more easily.

Given the relevance and significance of the file system to the operating system, it is extremely important as computer scientists to possess a strong understanding of how it functions. The purpose of this project was to allow us to gain a more in-depth knowledge of a file system's properties by experimenting and performing a series of measurements on a Unix based platform. To do this, it was necessary to use certain system calls, such as `read`, `fsync`, `write`, `open`, and `close`, to determine how long certain file system operations might take, and deduce what actions are being performed by it.

By using these system calls, several properties were learned about *hopper*, a host located at NMSU. It was found that this machine's `/dev/sda1` file system uses a block size of 4096 bytes, and that 14 blocks are prefetched into cache by the file

system. It was also determined that the cache size is approximately 14GiB, and that the file system uses an extent-based allocation method with an extent size of approximately between 83 and 84MiB.

## II. METHODOLOGY

This section will include information about the choice of platform for this experiment, the general methodology of the project, and the more detailed steps taken for each of the four experiments preformed.

### Platform

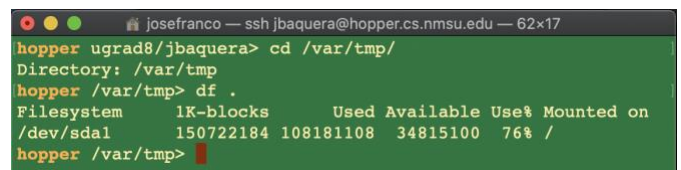
The openSUSE Leap Linux distribution was chosen as the UNIX-based platform to study. Such platform is sponsored by SUSE Linux GmbH and other companies and is widely used throughout the world. This particular Linux distribution is also the default Unix-based platform used at NMSU's computer science department. It is critical to note that the *hopper* local host was used to run all the experiments described in the following sections.

### General Methodology

Several methods were implemented and used to analyze different parts of the openSUSE Leap Linux distribution file system. To begin with, an 18GiB file was populated with random data and stored in the `/var/tmp/` directory using the following Linux-based command:

```
base64 /dev/urandom | head -c 18G > /var/tmp/Random_1
```

As Figure 1 demonstrates, this directory was chosen since it lives in `/dev/sda1`, which is a file system on *hopper* currently mounted on `/`. It is critical to note that any file that is stored in this directory will be written directly to local disk since such partition lives in the physical *sda* device. In addition, such methodology was used since measurements taken throughout the experiment needed to be measured on a local disk rather than on a distributed file system. This particular file is of extreme importance and its usage is explained in more detail in later sections.

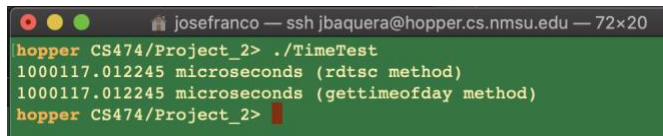


```
josefranco — ssh jbaquera@hopper.cs.nmsu.edu — 62x17
hopper ugrad8/jbaquera> cd /var/tmp/
Directory: /var/tmp
hopper /var/tmp> df .
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/sda1    150722184 108181108   34815100   76% /
hopper /var/tmp>
```

Figure 1 – Screenshot illustrating properties of the `/var/tmp/` directory.

## Timer Methodology

The aforementioned experiments used **rdtsc**, which is a highly accurate instruction that returns a 64-bit cycle count and is used



```
josefranco — ssh jbaquera@hopper.cs.nmsu.edu — 72x20
hopper CS474/Project_2> ./TimeTest
1000117.012245 microseconds (rdtsc method)
1000117.012245 microseconds (gettimeofday method)
hopper CS474/Project_2>
```

Figure 2 – Screenshot illustrating the accuracy of the **rdtsc** instruction.

to measure time. The accuracy and granularity of the timer used for these experiments was also tested since erroneous time measurements would lead to misleading and incorrect conclusions. Because of the importance of timer accuracy, the **gettimeofday** function was used to compare the results returned from **rdtsc**. As Figure 2 illustrates, both methods produced moderately accurate, but extremely precise, results when timing the **sleep(1)** instruction.

It is worth noting, however, that the cyclical nature of this counter has a tendency to produce some inconsistencies in the number of cycles it returns. For example, the instruction's reliance on hardware, such as processor speeds and power management, makes it produce different results from one machine to another. Another problem with **rdtsc** is that it is heavily dependent on the fact that a machine will, more than likely, have its processors have different start times. In order to reduce the effects of these inconsistencies in the data collected, each experiment was run multiple times. In other words, multiple trial runs of each experiment were performed in order to reduce experimental noise and variability within the collected data.

## Block Size Experiment Methodology

The first experiment performed for this project had the objective of determining the block size used by the file system to read data. To determine this, a small C program was written. The program takes an argument representing the number of bytes that will be read from the previously created *Random\_1* file in the */var/tmp* directory. The entered read size should preferably be a power of two since the size of a file system block will almost always be a power of two. This number is then used to allocate the buffer that will store the read data. It is worth noting that the buffer itself will not be used in any other way, except to read from the file.

The *Random\_1* file is opened and a **for** loop is employed to read the user inputted read size from the file a total of 100 times. The **rdtsc** timer is started before each execution of the **read** function, and then stopped immediately after the specified numbers of bytes have been read from the file. After the timer is stopped, it is checked that the actual number of bytes read matches the read size set by the user input, and the total elapsed time for the read is calculated and printed. For more information about the implementation of question 1, please refer to the *Question1.c* file.

For this project, the program was run two different times, one with input of 256 bytes, and one with 512 bytes. Once the program was done executing, all read time results were

recorded and graphed. It is worth mentioning that the recorded times were expected to be similar until the end of a block was reached, at which point the read time should increase in a significant manner. The block size can, therefore, be calculated by using the number of iterations that have been executed at the time of the read time increase and the number of inputted bytes.

## Prefetched Data Experiment Methodology

The objective of the second experiment in this project was to determine how much data is prefetched during the sequential read of a large file. The block size result produced by the first experiment was used as the read size in a short C program that was used to determine the size of the prefetched data. As with the previous experiment, a buffer was allocated using the block size. Again, it is worth mentioning that the buffer itself will not be used in any other way, except to be used in the **read** function.

The C program written for this experiment is very similar to the program written for the first experiment. The *Random\_1* file is opened and a **for** loop is used to read the file a total of 100 times. However, in this experiment, the file is read one block at a time, instead of a certain number of bytes, because it makes it simpler to determine how many blocks are prefetched by the file system. The **rdtsc** timer is started before each execution of the **read** function, and then stopped immediately after the block has been read. The program will then check that the actual number of bytes read matches the block size, and the total read time is calculated and printed. For more information about the implementation of question 2, please refer to the *Question2.c* file.

After executing the program, all read time results were recorded. As mentioned above, the program was run multiple times to calculate and graph the average results. It was expected that the recorded read times would be consistent until the end of the prefetched data is reached, at which point the read time should increase very significantly. The spike in read time allowed us to approximately calculate the number of blocks being prefetched into cache.

## Cache Size Experiment Methodology

The objective of the third experiment in this project was to determine the main memory cache size used by the file system. To determine this, a small C program was written. This program was coded to have a constant integer size of 268400000 to represent 268,400,000 bytes, or 256MiB, of data. This number is then used by the program to allocate the buffer that will store the read data. It is worth noting that the buffer itself will not be used in any other way, except to read from the *Random\_1* file stored in the */var/tmp/* directory.

The program will first employ a **for** loop with an additional two inner **for** loops. The outer **for** loop will iterate 84 times in order to read a total of 16GiB of random data from the *Random\_1* file (i.e. since the program is reading 256MiB at a time, and 16GiB = 16384MiB, then 16384/256 = 84).

The outer **for** loop will first open the *Random\_1* file and will run the first inner **for** loop. The first inner **for** loop will read 256MiB of random data a total of  $i + 1$  times. This pattern is illustrated in Table 1. It is worth mentioning that the **read** in this **for** loop will not be timed since this loop will, in essence, store the read bytes into main memory cache. A second inner **for** loop is needed to see if these bytes fit into cache. Such loop will be described in more detail further down in this section.

First Inner for Loop Algorithm	
Outer For Loop Iteration ( $i$ )	Bytes Read In Inner For Loop
0	256MiB
1	512MiB
.	.
.	.
63	16GiB

**Table 1** –Summary of algorithm used to read bytes in first inner for loop.

The program will close and reopen *Random\_1* in order to simulate “seeking” back to the beginning of the file. After the file is reopened, a second inner **for** loop will be run. This loop was implemented almost exactly as the first inner **for** loop (see Table 1), though the **rdtsc** timer is started before each execution of the **read** function, and then stopped immediately after the specified number of bytes have been read from the file. The time it took to read these bytes will be calculated and added to the average time after the timer is stopped. The program will then close the file in order to “seek” back during the next iteration of the outer **for** loop. Throughout execution, the program will check that the file is opened and closed correctly, and that the actual number of bytes read matches 256MiB. For more information about the implementation of question 3, please refer to the *Question3.c* file. It is critical to note that this method will not work for main memory cache sizes larger than 16GiB.

Before executing the program, the main memory cache was flushed multiple times with the following Linux-based command:

```
cat /var/tmp/Random_2 > /dev/null
```

It is worth mentioning that the *Random\_2* file is a 25GiB file populated with random data. Once the program finishes execution, all average elapsed time results are recorded and graphed. This graph is expected to stay uniform (i.e. flatlined) until the main memory cache is completely full, at which the read time should increase in a significant manner. This graph shape is expected since repeated reads to a group of bytes that fit in cache will be very fast while repeated reads to a group of bytes that do not fit in cache will be slow.

#### Allocation Method Experiment Methodology

The fourth and last experiment was meant to demonstrate the type of allocation method used by the file systems of the chosen host. The type of allocation method was determined before designing the experiment using the **df -Th** command on the */dev/sda1* device partition. Figure 3 illustrates that partition */dev/sda1* from the *hopper* host examined in this project uses

an extent-base4 file system. Therefore, it is important to note that the program described in this section will not work in systems that are not allocated with the extent-based method.

```
lilianaaguirresepaza — ssh lesparza@hopper.cs.nmsu.edu — 71x53
hopper ugrad4/lesparza> df -Th | grep "/dev/sda1"
/dev/sda1                ext4      144G  104G   34G  76% /
hopper ugrad4/lesparza>
```

**Figure 3** – Screenshot illustrating properties of the */var/tmp/* directory.

For this experiment, a C program similar to the ones described in experiments 1 and 2 was written. However, this experiment measures write syncing time, which is the time it takes to write the information written to a buffer to the disk, rather than the read time. Before the experiment was executed, a 16 GiB file named *LargeTestFile* was created in the */var/tmp* directory using the following command:

```
fallocate -l 16GB /var/tmp/LargeTestFile
```

The program defines a standard write size of 1 MiB or 1049000 bytes and uses this value to allocate a write buffer. It is worth noting that the **malloc** function allocates the buffer, but it does not empty it. Therefore, the buffer contains garbage when it is first allocated. Whatever the buffer contains when it is created is what will be written to the *LargeTestFile* since the content is not the focus of this experiment but the time it takes to write a specific number of bytes to the disk.

As in previous experiments, the file is opened and a **for** loop is established to iterate through 350 times. Within the loop, the buffer is first written to the file system cache. Then the timer is started and the data in the cache is synced to the disk space. After the sync is complete, the timer is stopped, and the total time elapsed for the sync is printed. For more information about the implementation of question 4, please refer to the *Question4.c* file.

The program was executed for a total of three trials, and for each trial, the sync time results were recorded and graphed. It is important to mention that the sync times at each iteration are expected to be similar in size until the end of the extent is reached, and the write pointer moves to the next extent. At this point, the sync time should show an increase, which allows us to calculate the size of the extents.

### III. RESULTS

This section will provide plots that will be used to address and answer the following four questions:

1. How big is the block size used by the file system to read data?
2. During a sequential read of a large file, how much data is prefetched by the file system?
3. How big is the main memory file cache?
4. For a system using an extent-based scheme, how big is the extent size?

Links to download the used data can be found in Appendix A.

## Block Size

Figure 4 provides a screenshot of the truncated code utilized to find the block size used by the file system.

```
/* Use a for loop that will read the desired number of bytes a set number of times. */
for( i = 0; i < 100; i++ ) {

    printf("RunNum%d", i);

    /* Start the cycle counter clock. */
    rStart = _rdtsc();

    /* Read the appropriate number of bytes from file. */
    sizeRead = read(fileDescriptor, buffer, readSizeUser);

    /* End the cycle counter clock. */
    rEnd = _rdtsc();

    /* Print the time it took to read the bytes (microseconds). */
    printf("%f\n", ((double) (rEnd - rStart)/constantConversion) *1000000);

} // end for.
```

Figure 4 – Screenshot of truncated Question1.c code.

In contrast, Figure 5 plots the average read time calculated from three different trial runs using a read size of 256 bytes, as well as the applicable iteration runs. As the plot illustrates, there are peaks at 1, 17, 33, 49, 65, 81, and 97. The distance between these peaks is always 16 iteration runs, or  $16 \times 256$  bytes = 4096 bytes = 4KiB. Therefore, a reasonable conclusion can be made that the block size used by the file system to read data is 4KiB.

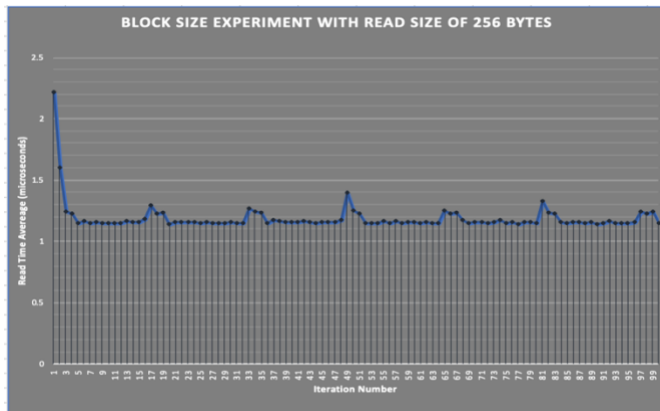


Figure 5 – Block size experiment plot with a read size of 256 bytes.

Figure 6, which plots the average read time calculated from three different trial runs using a read size of 512 bytes, as well as the applicable iteration runs, confirms the previously mentioned results. As this plot demonstrates, there are peaks at 1, 9, 17, 25, 33, 41, 49, 57, 65, 73, 81, 89, and 97. The distance between these peaks is always 8 iteration runs, or  $8 \times 512$  bytes = 4096 bytes = 4KiB.

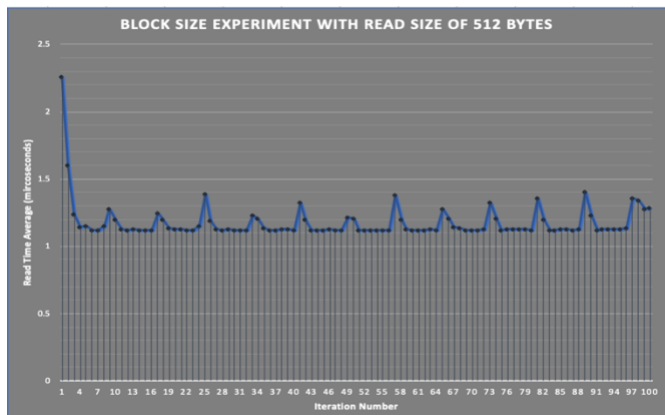


Figure 6 – Block size experiment plot with a read size of 512 bytes.

Figure 7 illustrates the output after running the following Linux-based command:

```
stat -fc %s .
```

As this screenshot demonstrates, the actual block size used by the file system is, indeed, 4096 bytes.

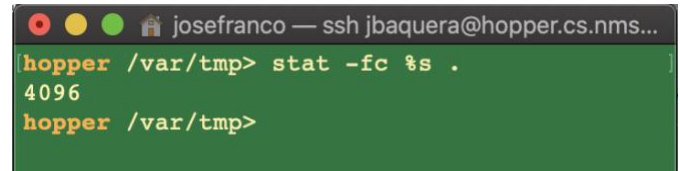


Figure 7 – Output after running the Linux-based command that can be used to find the block size used by the file system.

## Amount of Prefetched Data

Figure 8 below provides a screenshot of the truncated code utilized to find how much data is prefetched by the file system.

```
/* Use a for loop that will read 4096 bytes a set number of times. */
for( i = 0; i < 100; i++ ) {

    printf("RunNum%d", i);

    /* Start the cycle counter clock. */
    rStart = _rdtsc();

    /* Read the appropriate number of bytes from file. */
    sizeRead = read(fileDescriptor, buffer, readSize);

    /* End the cycle counter clock. */
    rEnd = _rdtsc();

    /* Print the time it took to read the bytes (microseconds). */
    printf("%f\n", ((double) (rEnd - rStart)/constantConversion) *1000000);

} // end for.

/* Free the allocated memory. */
free(buffer);
```

Figure 8 – Screenshot of truncated Question2.c code.

In contrast, Figure 9 plots the average read time calculated from three different trial runs using a read size of 1 block, as well as the applicable iteration runs. As the plot illustrates, there are peaks at 1, 15, 30, 45, 60, 75, and 90. From this pattern, it can be deduced that the file system fetches 15 blocks at one time, though it prefetches 14 blocks. This implies that every 15th block read will take longer. This deduction, however, is more of a hypothesis rather than a fact since the spike at iteration run 1 produces more questions than answers.

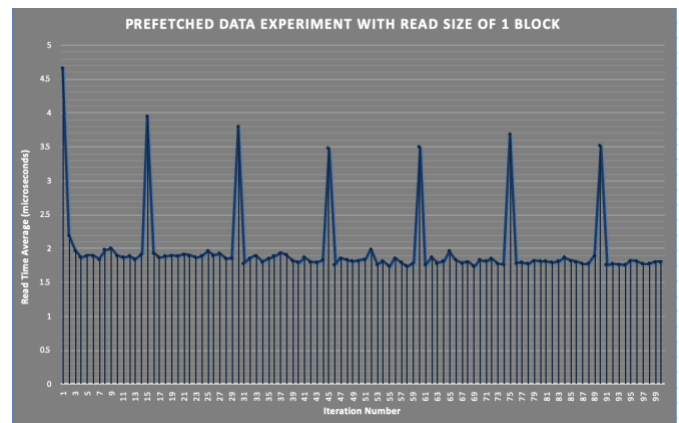


Figure 9 – Prefetched data experiment with read size of 1 block.



## Main Memory Cache Size

Figure 9 below provides a screenshot of the truncated code utilized to find main memory cache size.

```
/* For loop that will go until 16GiB are read. Since we are
 * reading 256MiB at a time, and 16GiB = 16384 MiB, then
 * 16384/256 = 64. */
for ( i = 0; i < 64; i++ ) {

    printf("RunNum%d", i);
    /* Open the 16GiB file in read only mode. Check that it opened successfully. */
    fileDescriptor = open(filename_1, O_RDONLY);

    /* For loop that will read 256MiB of data a certain number of times. This for loop
     * will, in essence, store the bytes read in cache.
     */
    for( k = 0; k < i+1; k++ ) {

        /* Read 256MiB of data. */
        sizeRead = read(fileDescriptor, buffer, readSize_1);

    } // end for.

    /* Close file. */
    close(fileDescriptor)
    /* Open the 16GiB file in read only mode again.*/
    fileDescriptor = open(filename_1, O_RDONLY);

    /* For loop that will read 256MiB of data a certain number of times. This for loop
     * will, in essence, be slower if these bytes are not in cache.
     */
    for( k = 0; k < i+1; k++ ) {

        /* Start the cycle counter clock. */
        rStart = _rdtsc();
        /* Read 256MiB of data. */
        sizeRead = read(fileDescriptor, buffer, readSize_1);
        /* End the cycle counter clock. */
        rEnd = _rdtsc();
        /* Keep track of the average time it took to read 256MiB of data. */
        average = average + (((double) (rEnd - rStart)/constantConversion) *1000000) / (i+1);

    } // end for.

    /* Print the average time it took to read the bytes and reset the average variable. */
    printf("%f\n", average);
    average = 0;
    /* Close file. */
    close(fileDescriptor);

} // end outer for.
```

Figure 9 – Screenshot of truncated Question3.c code.

In contrast, Figure 10 plots the average read time calculated from three different trial runs using a read size of 256MiB, as well as the applicable iteration runs. As the plot illustrates, the average read time begins to increase at the 53 iteration run, which implies that, at this point, some of the blocks that the program is trying to read are not in main memory and must be brought in from the local disk. Therefore, it can be concluded that the main memory cache size is around  $52 \times 256\text{MiB} = 13312\text{MiB}$ , or 13GiB.

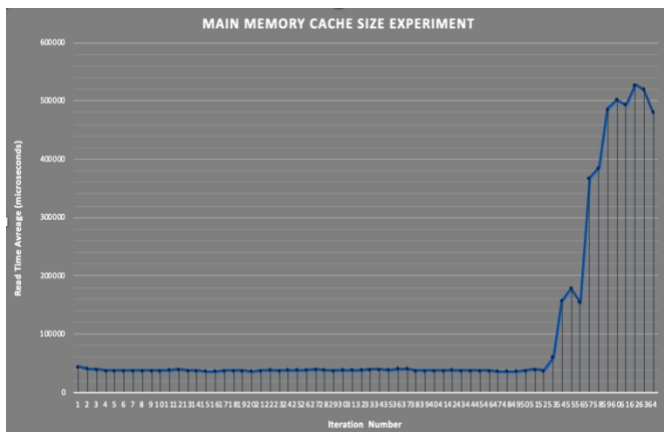


Figure 10 – Main memory cache experiment plot with a read size of 256MiB.

Figure 11 illustrates the output after running the following Linux-based command:

```
free -h
```

As this screenshot demonstrates, the actual main memory cache size is around 14GiB, which is close to the approximated

13GiB cache size mentioned in this section. The inferred cache size previously mentioned does not match perfectly since some of main memory is allocated to other critical processes and file system buffers.

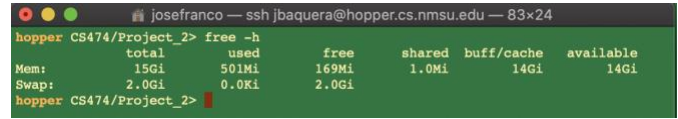


Figure 11 – Screenshot illustrating the output of the `free -h` Linux-based command.

## Allocation Method and Extent Size

As mentioned in a previous section, the allocation method used by *hopper*'s file system is extent-based. More specifically, the file system is type *ext4*. Figure 12 provides a screenshot of the truncated code utilized to find the extent size of the file system.

```
/* Use a for loop that will write 1 MiB of random data at a time to a large test file.*/
for( i = 0; i < 350; i++ ) {

    printf("RunNum%d", i);

    /* Write to file system cache. */
    write(fileDescriptor, buffer, writeSize );

    /* Start the cycle counter clock. */
    rStart = _rdtsc();
    /* Flush all data stored in buffers to actual disk space and check that it is done
     * successfully. */
    fsync(fileDescriptor);
    /* End the cycle counter clock. */
    rEnd = _rdtsc();

    /* Print the time it took to read the bytes (microseconds). */
    printf("%f\n", (((double) (rEnd - rStart)/constantConversion) *1000000));

} // end for.
```

Figure 12 – Screenshot of truncated Question4.c code.

In contrast, Figure 13 plots the `fsync` time from one trial run using a write size of 1MiB, as well as the applicable iteration runs. As the plot illustrates, there are peaks at 78, 162, 245, and 329. The distance between these peaks alternates between 83 and 84 iteration runs. Figures 14 and 15 (two different trial runs) also illustrate this similar pattern. Therefore, given that at each iteration the program is writing exactly 1MiB to disk, it can be inferred that the extent size used by the file system is between 83 and 84MiB.

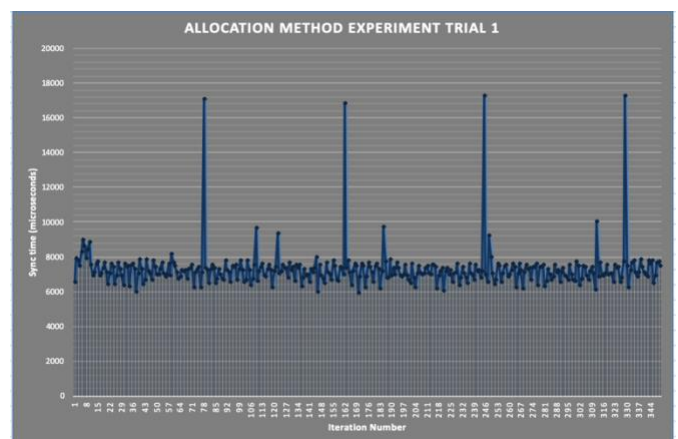


Figure 13 – Allocation method experiment plot for trial 1 with write size of 1MiB and 350 iterations.

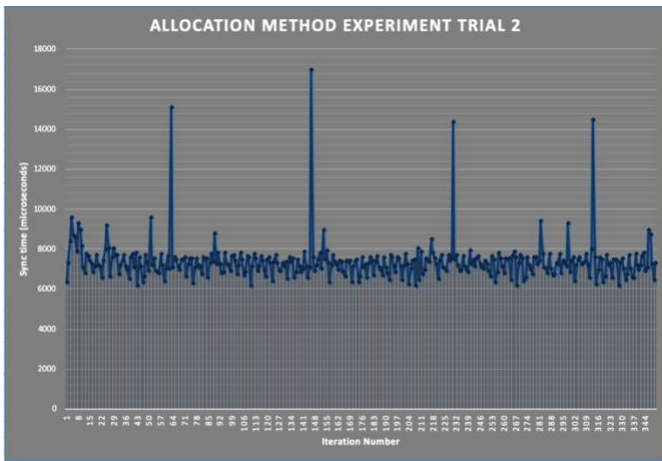


Figure 14 – Allocation method experiment plot for trial 2 with write size of 1MiB and 350 iterations.

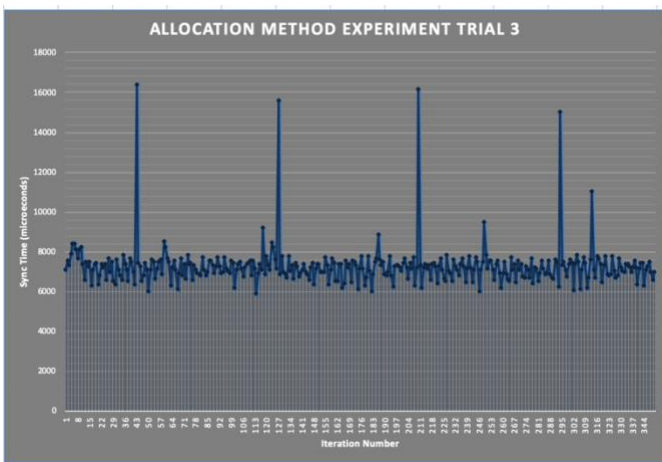


Figure 15 – Allocation method experiment plot for trial 3 with write size of 1MiB and 350 iterations.

This deduction, however, is more of a hypothesis rather than a fact since the alternation between 83 and 84MiB raises more questions than answers. Nevertheless, such numbers are reasonable since a single extent in ext4 can map up to 128MiB of contiguous space when using a 4KiB block size. If the inferred numbers were larger than 128MiB, then the results would be clearly erroneous. It can be deduced that *hopper* does not have free contiguous blocks that would provide 128MiB of space for each extent.

#### IV. CONCLUSIONS

This project has allowed us to gain a deeper understanding of a file system. More specifically, the experiments and calculations performed on *hopper* allowed us to determine some important properties of its file system and gain some insight into how it functions and aids the operating system. Through the aforementioned experiments, it was concluded that *hopper*'s `/dev/sda1` filesystem uses a block size of 4096

bytes, and that when a read occurs, 14 contiguous blocks of this size are prefetched into main memory cache. This allows for faster read times of these 14 blocks and avoids having to fetch a block from disk every time a read occurs, which would slow down the system significantly. It was also concluded that the main memory cache has a size of approximately 14GiB, and it aids the OS in ensuring faster read times since accessing files stored in cache is faster than accessing files stored in disk. Therefore, having a larger sized cache can be beneficial for system performance. Finally, it was concluded that *hopper*'s file system uses an extend-based allocation method, or ext4 to be more specific, and that each extent has a size of approximately between 83 and 84MiB.

One of the most important lessons, and something that should be considered anytime we work with a file system, is that, despite the findings of these experiments, each file system will yield different results. Not all file systems are created equal, and their implementation will vary based not only on the operating system, but on the hardware supporting it as well.

#### V. REFERENCES

[1] S. Abraham, G. Peter, and G. Greg, *Operating System Concepts*. Hoboken, NJ: John Wiley and Sons.

#### VI. APPENDIX A

The yielded data and resulting graphs can be accessed through the following links:

- Experiment 1 – [Question1\\_Data](#)
- Experiment 2 – [Question2\\_Data](#)
- Experiment 3 – [Question3\\_Data](#)
- Experiment 4 – [Question4\\_Data](#)