Global Journal of Computer Science & Technology
Volume 11 Issue 5 Version 1.0 April 2011
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
ISSN: 0975-5861

# IO Bound Property: A System Perspective Evaluation & Behavior Trace of File System

By Wasim Ahmad Bhat , S. M. K. Quadri

*Abstract-* : File systems have been mostly benchmarked as per the application perspective. This approach hides all the underlying complexities of the system including the actual I/O being done with the secondary storage device like magnetic disk. The IO bound property of a file system is necessarily to be evaluated because the most dominant performance limiting factor of a file system is its I/O operation with secondary storage device. This IO bound property of file system dictates the quantity and frequency of IO that a file system does with secondary storage device. In this paper, we argue system perspective of file system benchmarks and develop a benchmark to evaluate some common disk file systems for IO bound property. The goal of this paper is to better understand the behavior of file systems and unveil the low level complexities faced by file systems.

*Keywords:* *File System, IO Bound, Evaluation, Trace.*

*Classification:* *GJCST Classification: FOR Code: 080610,080402,080501*


*Strictly as per the compliance and regulations of:*

# IO Bound Property: A System Perspective Evaluation & Behavior Trace of File System

Wasim Ahmad Bhat[α], S. M. K. Quadri[Ω]

*Abstract-* File systems have been mostly benchmarked as per the application perspective. This approach hides all the underlying complexities of the system including the actual I/O being done with the secondary storage device like magnetic disk. The IO bound property of a file system is necessarily to be evaluated because the most dominant performance limiting factor of a file system is its I/O operation with secondary storage device. This IO bound property of file system dictates the quantity and frequency of IO that a file system does with secondary storage device. In this paper, we argue system perspective of file system benchmarks and develop a benchmark to evaluate some common disk file systems for IO bound property. The goal of this paper is to better understand the behavior of file systems and unveil the low level complexities faced by file systems.

*Keywords-* *File System, IO Bound, Evaluation, Trace.*

## I. Introduction

File system is an essential part of an operating system which dictates overall system performance and application specific performance. Thus, evaluating and analyzing file systems is necessary. There are many factors that affect the file system performance. Those factors include disk block organization, file name mapping, meta-data structure, reliability, concurrency control and data searching algorithms. Besides the above data storage related factors, the cache and memory buffer management scheme in the operating system plays a very important role in system I/O performance. Because the file system mitigates access to data on a mass storage subsystem, it has certain behavioral and functional characteristics that affect I/O performance from an application and/or system point of view. Measuring file system performance is significantly more complicated than that of the underlying disk subsystem because of the many types of higher-level operations that can be performed (allocations, deletions, directory searches, etc.)

*About[α] - Research scholar in P. G. Department of Computer Sciences, Kashmir University, India. He did his Bachelor's degree in Computer Applications from Islamia College of Science & Commerce and Master's degree in Computer Applications from Kashmir University.*
*E-mail- wasim.ahmed.bhat@gmail.com*
*About[Ω] - Head, P. G. Department of Computer Sciences, Kashmir University, India. He did his M. Tech. in Computer Applications from Indian School of Mines and Ph. D. in Computer Sciences from Kashmir University.*
*E-mail- quadrismk@hotmail.com*

Benchmarking file systems is a process of gathering some performance data by running a specific workload on a specific system. This technique clearly provides an accurate evaluation of performance of that system for that workload. Although file system design has advanced a lot, benchmarks for file system still lag far behind. The benchmarks used in file system research papers suffer from several problems. First, there is no standard benchmark. The closest to a standard is Andrew benchmark [1], but even then, some researchers use the original version while others use modified version [2][3]. Comparing results from different papers becomes difficult due to lack of standardization. Secondly, existing benchmarks are inadequate to measure file systems as they do not scale with technology [4], measure only part of file system [4][5][6] and do not yield results that would help a user to determine how a system might perform or would point designer towards possible areas for improvement. In addition to mentioned problems, file system benchmarks stress mainly on application perspective to evaluate and analyze the performance of a file system. Thus, this approach hides all the underlying complexities of system including the actual I/O being done with the secondary storage device like disk. This IO bound property of a file system dictates the quantity and frequency of I/O that a file system does with the secondary storage device to complete a particular operation. Hence, this property of file system is necessarily to be evaluated because the most dominant performance limiting factor of a file system is its I/O operation with secondary storage device. Although certain optimizations have been included in operating system like disk cache, read-ahead, delay-write, etc. to minimize the frequency of I/O being done by the file system, but the quantity of I/O is operation and design dependent. As such we need to evaluate and analyze file systems for the quantity of I/O being done with the secondary storage device for a set of different operations to look into the design efficiency of a file system. In other words, we need to analyze file system from system perspective and evaluate them for IO bound property.

In this paper, we evaluate and analyze 4 common file systems across WINDOWS and LINUX platforms for their IO bound property keeping all the system parameters for all operations constant across all the file systems under evaluation. The results so

obtained show that NTFS file system does lot of I/O with the disk. Also, LINUX file systems did least I/O with disk. Further, we observed 3 general patterns of disk accesses done by these file systems as far as our tests are concerned.

## II. Background and Related Work

We now review the basics of benchmarking file systems and present the work that is somehow related or point towards our concept.

Lucas [7] stated three reasons to obtain performance data: to know which system is better, how to improve its performance and how well will it perform. Thus, benchmarking assists customers looking to buy a better system and system designers looking for possible areas for improvement. Benchmarks may be categorized in two ways. One way is to categorize a benchmark as being either a synthetic or an application benchmark; the other way is as a macro- or micro-benchmark.

Application benchmarks consist of programs and utilities that a user can actually use like SPECint92 [8].

Synthetic benchmarks, on the other hand, model a workload by executing various operations in a mix consistent with the target workload like Bonnie [5].

Macro-benchmarks measure the entire system, and usually model some workload; they can be either synthetic or application benchmarks like IOStone [4].

Micro-benchmarks measure a specific part of a system. They can be thought of as a subset of synthetic benchmarks in that they are artificial; however, they do not try to model any real workload whatsoever. An example of a micro-benchmark is the create micro-benchmark from the original LFS paper: It timed how long the system took to create 10,000 files [9]. Micro-benchmarks are excellent for pointing out potential areas for improvement within the system as few operations are tested to isolate their specific overheads within the system. Thus, they measure specific part of file system. Generally, four parameters are the most common targets of file system micro-benchmarks:

1. The time to create a file,
2. The time to delete a file,
3. The throughput for reading files, and
4. The throughput for writing files.

Occasionally researchers use micro-benchmarks to measure other quantities, such as the time to create a symbolic link or read a directory. These quantities are measured less often, because the corresponding file system operations are perceived to occur less often in real file system workloads.

When an application makes a request to open, close, read, or write a file, the request is propagated through the operating system consisting of several levels of hierarchy before it reaches the actual storage media. These levels of hierarchy add optimizations by implementing a disk cache to cache the recently accessed disk blocks for anticipated use, buffer management scheme, merger read and write. This hierarchy tries to minimize the frequency of disk I/O by reading and then caching more blocks of disk than requested to anticipate a sequential read. The delayed write and caching of disk blocks tries to minimize the frequency of disk I/O by anticipating future updation and read of a disk block respectively whose write was requested. This can have a significant impact on both the meta-data and user data performance. This performance is further increased if the design of file system takes this optimization into consideration to minimize the quantity of disk I/O done for a particular operation.

Seltzer et al. [10] suggested that most benchmarks do not provide useful information as they are not designed to describe performance of a particular application. They argued for an application-directed approach to benchmarking, using performance metrics that reflect the expected behavior of a particular application across a range of hardware or software platforms. They proposed three approaches to application specific benchmarking: vector-based, trace-driven, and hybrid. Each methodology addresses a different set of benchmarking requirements and constraints. The fundamental principle behind vector-based performance analysis is the observation that in a typical computer system, each different primitive operation, whether at the application, operating system, or hardware level, takes a different amount of time to complete. Traditional benchmarking techniques ignore this fact and attempt to represent the overall performance of a computer system or subsystem as a scalar quantity. Vector-based techniques address this problem by representing the performance of underlying system abstractions as a vector quantity. Each component of this system characterization vector represents the performance of one underlying primitive, and is obtained by running an appropriate micro-benchmark.

Chen [6] laid out criteria for evaluating I/O systems which can be adapted for file systems as well. Chen states that an I/O benchmark should be:

1. Prescriptive: It should point system designers towards possible areas for improvement.
2. I/O bound: The benchmark should measure the I/O system and not, for example, the CPU.
3. Scalable with advancing technology.
4. Comparable between different systems.
5. General: Applicable to a wide variety of workloads.
6. Tightly specified: No loopholes; clarity in what needs to be reported.

Tang [11] argued that these criteria should be applied to most benchmarking methodologies. He introduced a benchmark called *dtangbm*. This benchmark consisted of suite of micro-benchmarks called *fsbench* and a workload characterizer. The Phase I of *fsbench* measures disk performance so that it can be known whether improvements are due to disk or file system and can be compared to Phase III which measures file system block allocation policy to determine what overhead the file system imposes.

Traeger et al. [12] argued that some guidelines be followed while designing a micro benchmark. The two underlying themes of those guidelines are as follows:

1. *Explain What Was Done in as Much Detail as Possible.* This can help others understand and validate the results.
2. *In Addition to Saying What Was Done, Say Why It Was Done That Way?*

Ruwart [13] argued that not only are benchmarks ill-suited for testing but will fare even worse in future because of systems complexities. He presented the point of view from which the performance is measured. Three of the more generally accepted perspectives are:

1. Application
2. System
3. Storage Subsystem

The Application perspective is what most of the file system benchmarks represents. From this perspective all of the underlying system services and hardware functions are hidden. This perspective includes all the cumulative effects of other applications running at the same time as the benchmark run. This is also true for applications running on other machines that may be simultaneously accessing the storage subsystem under test. From this perspective the results of a benchmark can be skewed due to undesirable interactions from these other applications and other machines. The Application perspective can also divide I/O operations into the two distinct categories (Meta data and User data) based on the type of higher-level operation being performed. The Application interface to the file system is generally through high-level system calls such as open, close, read, write, and create. There are also higher level system calls that perform such operations as rename, create directory, remove, and lookup a name. It is the performance of these operations that ultimately determine the overall performance that the application sees for both metadata and user data operations.

The System perspective is viewed by running system-monitoring tools during a benchmark run. These tools provide coarse-grained real-time monitoring of the system I/O activity for such high-level operations as file reads and writes as well as the number of operations actually sent to the storage subsystem on a device-by-device basis. From this perspective it is possible to see and measure the effect of other applications that are running concurrently with the benchmark program. Furthermore, with some of the more sophisticated system monitoring tools, it is possible to monitor the activity on other systems that may be sharing access to the storage subsystem under test. However, there is still a problem with getting a complete view of all the systems on a common reference clock in order to better understand the interaction of all the systems with the shared storage subsystem.

The Storage Subsystem perspective is the most difficult to monitor since there are not many tools available to collect performance data from the storage subsystem.

## III. IO Bound Property

The smallest addressable read and write unit of secondary storage like disk is Sector. Typically sector size is 512 bytes. Although, disk drives allow random read and write of individual sectors, for performance reasons, file systems prefer to read and write a sequence of consecutive sectors called Cluster. Thus, the smallest addressable read and write unit of a file system is cluster. Cluster sizes vary from one sector to many in size. Clusters reduce the frequency of I/O operations by reading and writing more than one sector sequentially at a time which would have otherwise cost many individual reads and writes. File systems vary greatly in cluster sizes, allocation and layout policies, in addition to other parameters.

Linux operating system maintains an in-memory disk cache of recently accessed disk blocks in a hope that these blocks will be accessed again [14]. The blocks correspond to individual sectors of disk. When a process issues a file system syscall, the call passes through a hierarchy of layers within operating system and finally reaches the file system drives. The file system converts the call into appropriate disk blocks to be read or written as per the design of the file system mounted. Instead of directly reading and writing the disk block, it checks the disk cache for the block. If it is found, the block is accessed from cache and hence saving an I/O operation. If it is not found, the block is read from disk and cached. Linux further optimizes the I/O performance by asynchronously reading ahead few blocks in anticipation of sequential access whenever a block is to be read. It also delays write of updated blocks in anticipation of further updation of same block whenever a block is to be written or updated. All these optimizations by Linux operating system are done to reduce the number of I/O operations directly done with disk because an I/O operation with disk is costlier in time than with disk cache.

But due to design diversities in file systems, varying size of clusters, different allocation and layout

policies these optimizations are exploited by different file systems up to different levels. The file system benchmarks stress mainly on application perspective to evaluate and analyze a file system. This way they analyze the amount of data read and written by an application at higher level without being concerned about whether the data was read or written from cache only, disk only or partly from cache and partly from disk. Further, they ignore the data other than user data read or written by the application like when an application tries to open a file for reading they ignore the number of disk blocks that might have been read to locate the file on the mounted volume. Hence, this perspective does not give the actual measure of the IO bound property of a file system.

An IO bound property of a file system means the quantity and frequency of I/O operations that a file system does with the secondary storage device to complete a particular operation. Hence, this property of file system is necessarily to be evaluated because the most dominant performance limiting factor of a file system is its I/O operation with secondary storage device.

## IV. Performance Monitoring Tools

Two issues are to be considered when collecting performance data; the type of data to be collected and when to collect it. Concerning the first issue, there is essentially one type of data to collect: Number of disk block read and written from disk. The second issue of when to collect the data is obviously to be done after the completion of every individual workload generator.

Linux operating system provides many utilities to gather statistics about system resources used by the benchmark tests. Many utilities come along the default Linux package while others can be downloaded for free and recompiled for the distribution. We will review the most common, popular and useful utilities for system resource monitoring in Linux.

VMSTAT [15] reports information about processes, memory, paging, block I/O, traps, and cpu activity. The first report produced gives averages since the last reboot. Additional reports give information on a sampling period of length delay. The process and memory reports are instantaneous in either case. For Disk Mode, it reports total reads completed successfully, grouped reads (resulting in one I/O), sectors read successfully, milliseconds spent reading, total writes completed successfully, grouped writes (resulting in one I/O), sectors written successfully, milliseconds spent writing. For Disk Partition Mode it reports total number of reads issued to this partition, total read sectors for partition, total number of writes issued to this partition, total number of write requests made for partition.

IOSTAT [16] reports Central Processing Unit (CPU) statistics and input/output statistics for devices and partitions. The *iostat* command is used for monitoring system input/output device loading by observing the time the devices are active in relation to their average transfer rates. The *iostat* command generates reports that can be used to change system configuration to better balance the input/output load between physical disks. The first report generated by the *iostat* command provides statistics concerning the time since the system was booted. Each subsequent report covers the time since the previous report. All statistics are reported each time the *iostat* command is run. The report consists of a CPU header row followed by a row of CPU statistics. The second report generated by the *iostat* command is the Device Utilization Report. The device report provides statistics on a per physical device or partition basis. The device report generated constitutes of fields that indicate:

1. The number of transfers per second that were issued to the device. A transfer is an I/O request to the device. Multiple logical requests can be combined into a single I/O request to the device. A transfer is of indeterminate size.
2. The amount of data read from the device expressed in a number of blocks per second. Blocks are equivalent to sectors with 2.4 kernels and newer, and therefore have a size of 512 bytes. With older kernels, a block is of indeterminate size.
3. The amount of data written to the device expressed in a number of blocks per second.
4. The total number of blocks read.
5. The total number of blocks written.
6. The amount of data read from the device expressed in kilobytes per second.
7. The amount of data written to the device expressed in kilobytes per second.
8. The total number of kilobytes read.
9. The total number of kilobytes written.

SAR [17] is used to collect, report or save system activity information. The sar command writes to standard output the contents of selected cumulative activity counters in the operating system. The accounting system, based on the values in the count and interval parameters, writes information the specified number of times spaced at the specified intervals in seconds. The device report generated constitutes of fields that indicate:

1. The number of transfers per second that were issued to the device. Multiple logical requests can be combined into a single I/O request to the device. A transfer is of indeterminate size.
2. Number of sectors read from the device. The size of a sector is 512 bytes.

3. Number of sectors written to the device. The size of a sector is 512 bytes.

**IOTOP** [18] watches I/O usage information output by the Linux kernel (requires 2.6.20 or later) and displays a table of current I/O usage by processes or threads on the system. At least the CONFIG_TASK_DELAY_ACCT and CONFIG_TASK_IO_ACCOUNTING options need to be enabled in Linux kernel build configuration, these options depend on CONFIG_TASKSTATS. *iotop* displays columns for the I/O bandwidth read and written by each process/thread during the sampling period. It also displays the percentage of time the thread/process spent while swapping in and while waiting on I/O. For each process, its I/O priority (class/level) is shown. In addition, the total I/O bandwidth read and written during the sampling period is displayed at the top of the interface.

**PIDSTAT** [19] command is used for monitoring individual tasks currently being managed by the Linux kernel. It writes to standard output activities for every task selected with option -p or for every task managed by the Linux kernel if option -p ALL has been used. Not selecting any tasks is equivalent to specifying -p ALL but only active tasks (tasks with non-zero statistics values) will appear in the report. The *pidstat* command can also be used for monitoring the child processes of selected tasks. The interval parameter specifies the amount of time in seconds between each report. A value of 0 (or no parameters at all) indicates that tasks statistics are to be reported for the time since system startup. The count parameter can be specified in conjunction with the interval parameter if this one is not set to zero. The value of count determines the number of reports generated at interval seconds apart. If the interval parameter is specified without the count parameter, the pidstat command generates reports continuously.

**COLLECTL** [20] utility is a system monitoring tool that records or displays specific operating system data for one or more sets of subsystems. Any set of the subsystems, such as CPU, Disks, Memory or Sockets can be included in or excluded from data collection. Data can either be displayed back to the terminal, or stored in either a compressed or uncompressed data file. The data files themselves can either be in raw format or in a space separated plottable format such that it can be easily plotted using tools such as *gnuplot* [21] or *excel* [22]. Data files can be read and manipulated from the command line, or through use of command scripts.

**BLKTRACE** [23] is a block layer I/O tracing mechanism which provides detailed information about request queue operations up to user space. There are three major components: a kernel component, a utility to record the I/O trace information for the kernel to user space, and utilities to analyze and view the trace information. *blktrace* receives data from the kernel in buffers passed up through the debug file system. Each device being traced has a file created in the mounted directory for the *debugfs* [24], which defaults to /sys/kernel/debug.

**PROC** [25] file system is a pseudo-file system which is used as an interface to kernel data structures. It is commonly mounted at /proc. Most of it is read-only, but some files allow kernel variables to be changed. /proc/sys/vm/ directory facilitates the configuration of the Linux kernel's virtual memory (VM) subsystem. The kernel makes extensive and intelligent use of virtual memory, which is commonly referred to as swap space. The /proc/sys/vm/block_dump file configures block I/O debugging when enabled. All read/write and block dirtying operations done to files are logged accordingly. This can be useful if diagnosing disk spin up and spin downs for laptop battery conservation. All output, when block_dump is enabled, can be retrieved via *dmesg* [26]. The default value is 0 and can be enabled by setting its value to 1.

## V. The Benchmark

The most crucial part for evaluating and analyzing file systems for their IO bound property is to choose the set of tests that are to be executed which will give us some insight of the IO bound property. To make the choice simple and logical, we tried to find out the types of file system operations that vary in the quantity of I/O being done with the disk for different file systems due to their design variations. The ruled out option is, thus, a large file where the user data dominates the disk I/O and this dominance is constant throughout the file systems under evaluation. This makes one criterion clear; we are going to test large number of empty or small sized files.

We identified following file systems operations in which the quantity of disk I/O varies greatly for different file systems due to their design.

| Test Number | Description | Corresponding Figure Set |
|---|---|---|
| Test 1 | Create 10,000 files with 'touch' utility | Figure 1 |
| Test 2 | Run 'find' utility on that directory | Figure 2 |
| Test 3 | Remove these 10,000 files using 'rm' utility | Figure 3 |
| Test 4 | Create 10,000 directories with 'mkdir' utility | Figure 4 |
| Test 5 | Run 'find' utility on that directory | Figure 5 |
| Test 6 | Remove these 10,000 directories using 'rm' utility | Figure 6 |

The tests to be performed are listed as per the sequence they are executed along necessary Linux utility used and figure set that depicts their sector traces. The test code is organized as a shell script. All file systems are to be created using default options and each file system is tested on a cleanly made file system. All tests are to be run 3 times and the average is to be taken. Between every test; cache is flushed. The test measures quantity of I/O in terms of sectors read or written from the disk for every individual test ignoring the requests fulfilled by the cache and the time consumed to complete the operation.

Before the file system under evaluation is mounted we enable block dumping in kernel by setting /proc/sys/vm/block_dump to 1. After this, if any read or write request has to do disk I/O and as such is not fulfilled by cache, the corresponding operation, device file and block is dumped. The interested data can be collected by using 'dmesg –c' which gets the dumped operations and clears the log. The collection can be further refined by piping the output of 'dmesg –c' to 'grep sda1' (say) to get all the operations pertaining to some mounted file system corresponding to device file sda1.

## VI. File Systems To be Evaluated

We evaluate the performance of four file systems, two viz. FAT32 and NTFS being Windows native and two viz. Ext2, and Ext3 being Linux native. We test them under Fedora Core Release 7 (Moonshine) Kernel 2.6.21. To make the paper self-contained, we briefly describe the tested file systems as follows:

a) *FAT32*

The FAT (File Allocation Table) file system was developed in the late 1970s and early 1980s and was the file system supported by the Microsoft® MS-DOS® operating system [27]. FAT was originally developed for floppy disk drives less than 500K in size. As storage capacity increased, FAT was enhanced to support large storage media. As such we have three fully documented FAT file system types: FAT12, FAT16 and FAT32. FAT32, which can address large storage media and is supported by all major desktop operating systems, is still the most widely used file system in portable digital devices [28]. As compared to other file systems, the performance of FAT is poor as it uses simple data structures, making file operations time-consuming and inefficient disk space utilization in situations where many small files are present. But for same simple design and legacy it is supported by almost all existing operating systems for personal computers. This makes it a useful format for solid-state memory cards and a convenient way to share data between different operating systems. exFAT [29] is the recent compilation of Microsoft® while KFAT [30], TFAT [31] and FATTY [32] are the reliability enhancements to the actual design by the same and other researchers.

b) *NTFS*

The New Technology File System [33] was originally developed for Windows NT and now is used in Windows NT, 2000, XP, Vista and 7. NTFS provides performance, reliability, and functionality not found in FAT design. NTFS includes security and access controls, encryption support, and has reliability control built in, in the form of a journaling file system. In NTFS, all file data—file name, creation date, access permissions, and contents—are stored as metadata in the Master File Table. NTFS allows any sequence of 16-bit values for name encoding (file names, stream names, index names, etc.). NTFS contains several files which define and organize the file system. In all respects, most of these files are structured like any other user file ($Volume being the most peculiar), but are not of direct interest to file system clients. These metafiles define files, back up critical file system data, buffer file system changes, manage free space allocation, satisfy BIOS expectations, track bad allocation units, and store security and disk space usage information. NTFS includes several new features over its predecessors: sparse file support, disk usage quotas, reparse points, distributed link tracking, and file-level encryption, also known as the Encrypting File System (EFS).

c) *Ext2*

The Second Extended File System was designed and implemented to fix some problems present in the first Extended File System. The goal was to provide a powerful file system, which implements UNIX file semantics and offers advanced features. The Second Extended file system is the default file system in most Linux distributions [34] and is the most popular file system for Linux. In addition to the standard UNIX features, Ext2fs supports some extensions which are not usually present in UNIX file systems. File attributes allow the users to modify the kernel behavior when acting on a set of files. One can set attributes on a file or on a directory. In the latter case, new files created in the directory inherit these attributes. Ext2fs implements fast symbolic links. A fast symbolic link does not use any data block on the files system. The target name is not stored in a data block but in the I-node itself. This policy can save some disk space (no data block needs to be allocated) and speeds up link operations (there is no need to read a data block when accessing such a link). Ext2 borrows ideas from previous UNIX file systems using I-nodes to represent files and objects. It was designed to be extensible to make it possible to add features like journaling on at a later time.

d) *Ext3*

The Third Extended file system is a journaling file system developed by Stephen Tweedie [35] as an extension to Ext2. It is mount compatible to Ext2 file system, but includes a journaling file to provide recovery capability. Ext3 can use all of the existing applications that have already been developed to manipulate the Ext2 file system. Journaling increases the file system reliability, and reduces recovery time by eliminating the need for some consistency checks. The ext3 file system adds, over its predecessor: A Journaling file system, Online file system growth, Htree [36] indexing for larger directories (An HTree is a specialized version of a B-tree). Without these, any ext3 file system is also a valid ext2 file system. This has allowed well-tested and mature file system maintenance utilities for maintaining and repairing ext2 file systems to also be used with ext3 without major changes. Since ext3 aims to be backward compatible with the earlier ext2, many of the on-disk structures are similar to those of ext2. Because of that, ext3 lacks a number of features of more recent designs, such as extents, dynamic allocation of I-nodes, and block sub-allocation. There is no support of deleted file recovery in file system design. Ext3 driver actively deletes files by wiping file I-nodes [37] for crash safety reasons. That is why an accidental *'rm -rf *'* may cause permanent data loss. An enhanced version of the file system was announced by Theodore Ts'o [38] on June 28, 2006 under the name of ext4.

## VII. Experiment

The tests are performed on a clean 5GB file system containing nothing. The same partition is formatted to support all the file systems under evaluation in order to approximate disk latency. The computer used for testing is a PC with Intel Pentium 4 2.4 GHz CPU and 512MB DDR2 333MHz RAM. The hard drive is a 5400RPM 80GB ATA Device. The hard drive is partitioned into a 5GB partition to house the file system under evaluation and a 10GB partition for Fedora Core Release 7 (Moonshine) Kernel 2.6.21 [39] on an i386.

## VIII. Result & Discussion

Table 1 lists the result of all tests conducted and the quantity of I/O in terms of sectors read from disk by each individual test for each file system under evaluation. It can be observed from the table that NTFS file system did lot of I/O with disk by reading lot of sectors in total for all tests while ext3 did least I/O with disk reading least sectors in total for all tests. Also, there is comparatively a consistency in NTFS regarding the number of sectors read for each individual test.

*Table1. Result of tests (Sectors Read)*

| File System | FAT32 | NTFS | Ext2 | Ext3 |
|---|---|---|---|---|
| Test 1 | 19 | 1541 | 319 | 10 |
| Test 2 | 643 | 1566 | 35 | 9 |
| Test 3 | 643 | 4522 | 359 | 375 |
| Test 4 | 721 | 4128 | 559 | 214 |
| Test 5 | 643 | 1566 | 35 | 9 |
| Test 6 | 160721 | 4522 | 10569 | 10592 |

Table 2 lists the result of all tests conducted and the quantity of I/O in terms of sectors written to disk by each individual test for each file system under evaluation. It can be observed from the table that in tests where files where created and deleted, NTFS file system did lot of I/O with disk by writing lot of sectors while ext2 did least; whereas in test where directories where created FAT32 file system did lot of I/O with disk by writing a lot of sectors while NTFS file system did least I/O. In contrast in test where directories where deleted ext3 file system did lot of I/O with disk by writing lot of sectors while FAT32 did least I/O with disk. Again, it can be clearly observed that there is comparatively a consistency in NTFS file system regarding the number of sectors written for each individual test.

*Table2. Result of tests (Sectors Written)*

| File System | FAT32 | NTFS | Ext2 | Ext3 |
|---|---|---|---|---|
| Test 1 | 1210 | 2982 | 356 | 1171 |
| Test 2 | 0 | 0 | 2 | 2 |
| Test 3 | 627 | 3218 | 341 | 1403 |
| Test 4 | 34707 | 2980 | 11574 | 24980 |
| Test 5 | 0 | 0 | 2 | 2 |
| Test 6 | 790 | 2964 | 1244 | 9507 |

Further, the trace of all file systems under evaluation depicting the order in which the sectors on a track are accessed is shown in figure 1 to figure 6. The figures only show the position/location of sector on track that is being accessed (read/written) and thus can be used to interpret only rotational delay incurred in the tests and as such ignores the seek time.

The figure 1 depicts the trace of test 1 conducted on all file systems which creates 10,000 empty files. It is clear from the figure 1.1 that NTFS does a consistent sequential read while ext2 read every 8$^{th}$ sector of the track. Also, a consistent sequential write as depicted by figure 1.2 is done by FAT32 while NTFS, ext2 & ext3 did write every 8$^{th}$ sector of the track. From this figure, it can be comprehended that the trace of the file system in which every 8$^{th}$ sector of track is read or

written may efficiently utilizing the cache as LINUX does a sequential block read/write of 8 sectors [34].

The figure 2 depicts the trace of test 2 conducted on all file systems which finds a fugitive filename among the files created after flushing the cache. It is clear from the figure that both FAT32 & NTFS did a consistent sequential read.

The figure 3 depicts the trace of test 3 conducted on all file systems which deletes the 10,000 files created in test 1 after flushing the cache. It is clear from the figure 3.1 that both FAT32 & NTFS does a consistent sequential read while ext2 & ext3 read every 8th sector of track. Also, this pattern is repeated by FAT32 while writing sector whereas NTFS, ext2 and ext3 read every 8th sector of the track as shown in figure 3.2.

The figure 4 depicts the trace of test 4 conducted on all file systems which creates 10,000 empty directories after flushing the cache. It is clear from the figure 4.1 that both FAT32 & NTFS did a sequential read while ext2 & ext3 did a random read. Also, FAT32 repeated it pattern of sequential write and ext2 repeated its random write while writing sectors whereas NTFS & ext3 wrote every 8th sector of track as depicted by figure 4.2.

The figure 5 depicts the trace of test 5 conducted on all file systems which finds a fugitive directory name among the directories created after flushing the cache. It is clear from the figure that a sequential read is done by FAT32 & NTFS.

The figure 6 depicts the trace of test 6 conducted on all file systems which deletes the 10,000 directories created in test 4. It is clear from the figure 6.1 that both FAT32 & NTFS did a consistent sequential read while ext3 did a random read. Also, ext2 did a sequential read in different bands. Further, it is clear from figure 6.2 that FAT32 did a consistent sequential write while ext2 did a random write whereas both NTFS & ext3 did wrote every 8th sector of track.

From the discussion, we can conclude that NTFS is highly I/O bound so far as these tests are concerned. Also, LINUX file systems are less I/O bound. One important thing worth noting is the pattern in which the sectors are accessed on a track because this pattern will dictate the rotational delay incurred by these file systems to complete these operations. Generally, we observed only 3 types of patterns; Sequential, Random & every 8th sector. Among the observed patterns, LINUX file systems mostly exhibit a pattern in which every 8th sector is read or written and occasionally exhibited random and sequential behavior. Also, WINDOWS file systems exhibited both sequential and every 8th sector behavior with NTFS being highly I/O bound.

## IX. Conclusion

In this paper, we argued that current file system benchmarks mostly concentrate on application perspective and generally ignore the system perspective of benchmarking. We stressed on system perspective in addition to application perspective and presented IO bound property of file system. Then, we developed certain tests that will evaluate file systems for IO bound property and discussed various mechanisms to monitor the performance. Finally, we evaluated some common disk file systems across WINDOWS and LINUX platforms for this IO bound property. From the results we gathered, it can be summed up that NTFS does lot of I/O with disk and thus is highly IO Bound. At the same time, "LINUX file systems did least I/O with the and thus are least IO Bound Further, we observed 3 general disk access patterns;Random, Sequential and every 8th sector read/write. Among these patterns, we found that random pattern is occasionally exhibited by any file system while LINUX file systems mostly exhibit every 8th sector read/write while WINDOWS file systems either exhibit sequential or every 8th sector read/write behavior.

## X. Future Work

The traces gathered from the tests conducted are interesting. The traces although take into consideration only the rotational delay incurred, can tell us lot about the design efficiency of the file system. Thus, more investigations need to be done to understand the traces and correlate them with the design evaluation of file system. Further, the tests conducted here need to be broadened both in terms of number of tests and types of tests in addition to the platform on which the tests are conducted.

## References Références Referencias


1. Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N. and West, M.J. (1988), "Scale and Performance in a Distributed File System", ACM Transactions on Computer Systems, February 1988, 51-81.
2. Ousterhout, J.K. (1990), "Why Aren't Operating Systems Getting Faster As Fast As Hardware?", Proceedings of the 1990 USENIX Summer Technical Conference, June 1990, 247-256.
3. Seltzer, M.I., Smith, K., Balakrishnan, H., Chang, J., McMains S. and Padmanabhan, V. (1995), "File System Logging versus Clustering: A Performance Evaluation", Proceeding of the 1995 USENIX Technical Conference, 249-264.
4. Park, A. and Becker, J.C. (1990), "IOStone: A Synthetic File System Benchmark", Computer Architecture News 18, 2, 45-52.

5. Bray, T. (1990), Bonnie source code, netnews posting.
6. Chen, P.M. and Patterson, D.A. (1992), "A New Approach to I/O Benchmarks Adaptive Evaluation, Predicted Performance", UCB/Computer Science Dept. 92/679, University of California at Berkeley.
7. Lucas, H.C. (1972), "Performance Evaluation and Monitoring", Computing Surveys, September 1972, 79-91.
8. Standard Performance Evaluation Corporation. http://www.spec.org
9. Rosenblum, M. and Ousterhout, J.K. (1992), "The Design and Implementation of a Log-Structured File System", ACM Transactions on Computer Systems, February 1992.
10. Seltzer, M.I., Krinsky, D., Smith, A.K. and Zhang, A. (1999), "The Case for Application-Specific Benchmarking", In Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS). ACM, Rio Rica, AZ, 102–107.
11. Tang, D. (1995), "Benchmarking Filesystems", Techinal Report. TR-19-95, Harvard University.
12. Traeger, A., Zadok, E., Joukov, N. and Wright, C.P. (2008), "A Nine Year Study of File System and Storage Benchmarking", ACM Transactions on Storage.
13. Ruwart, T.M. (2001), "File system performance benchmarks, then, now, and tomorrow", In Proceedings of the 14th IEEE Symposium on Mass Storage Systems. IEEE, San Diego, CA.
14. Bovet, D.P. and Cesati, M., "Understanding the Linux Kernel", O'Reilly
15. Vmstat, http://linux.die.net/man/8/vmstat
16. Iostat, http://linux.die.net/man/1/iostat
17. Sar, http://linux.die.net/man/1/sar
18. Iotop, http://guichaz.free.fr/iotop/
19. Pidstat, http://man.he.net/man1/pidstat
20. Collect, http://collectl.sourceforge.net/
21. Gnuplot, http://www.gnuplot.info
22. Excel, http://office.microsoft.com/en-us/excel/
23. Blktrace, http://linux.die.net/man/8/blktrace
24. Debugfs, http://linux.die.net/man/8/debugfs
25. Proc, http://linux.die.net/man/5/proc
26. Dmesg, http://linuxgazette.net/issue59/nazario.html
27. Microsoft Corporation, (2000), "FAT32 File System Specification", http://microsoft.com/whdc/system/platform/firmware/fatgen.mspx
28. Bhat, W.A. and Quadri, S.M.K. (2009), "Review of FAT Data Structure of FAT32 file system", Oriental Journal of Computer Science & Technology, Volume 3, No 1
29. Microsoft Corporation, (2007), "Extended FAT File System", http://msdn2.microsoft.com/en-us/library/aa914353.aspx
30. Kwon, M.S., Bae, S.H., Jung, S.S., Seo, D.Y. and Kim, C.K. (2005), "KFAT: Log-based Transactional FAT File system for Embedded Mobile Systems", In Proceedings of 2005 US-Korea Conference, ZCTS-142.
31. Microsoft Corporation, (2007), "Transaction-Safe FAT File System", http://msdn2.microsoft.c0m/en-us/library/aa911939.aspx
32. Alei, L., Kejia, L. and Xiaoyong, L. (2007), "FATTY: A reliable FAT File System", Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, Pages: 390-395.
33. Duncan, R. (1989), "Design goals and implementation of the new High Performance File System", Microsoft Systems Journal.
34. Bovet, D.P. and Cesati, M. "Understanding the Linux Kernel", O'Reilly, ISBN 0-596-00565-2
35. Tweedie, S. (1998), "Journaling the Linux Ext2fs Filesystem", LinuxExpo '98.
36. Saynez, A. S., Somodevilla, M. J., Ortiz, M. M., Pineda, I. H. (2007), "H-Tree: A data structure for fast path-retrieval in rooted trees", Eighth Mexican International Conference on Current Trends in Computer Science (ENC 2007), pp.25-32.
37. Linux ext3 FAQ, http://batleth.sapienti-sat.org/projects/FAQs/ext3-faq.html
38. Ts'o, T. (2006), "Proposal and plan for ext2/3 future development work". Linux kernel mailing list. http://lkml.org/lkml/2006/6/28/454
39. http://fedoraproject.org/wiki/Releases/7

*Figure 1.* *Sector Read/Write Trace for Test 1.*

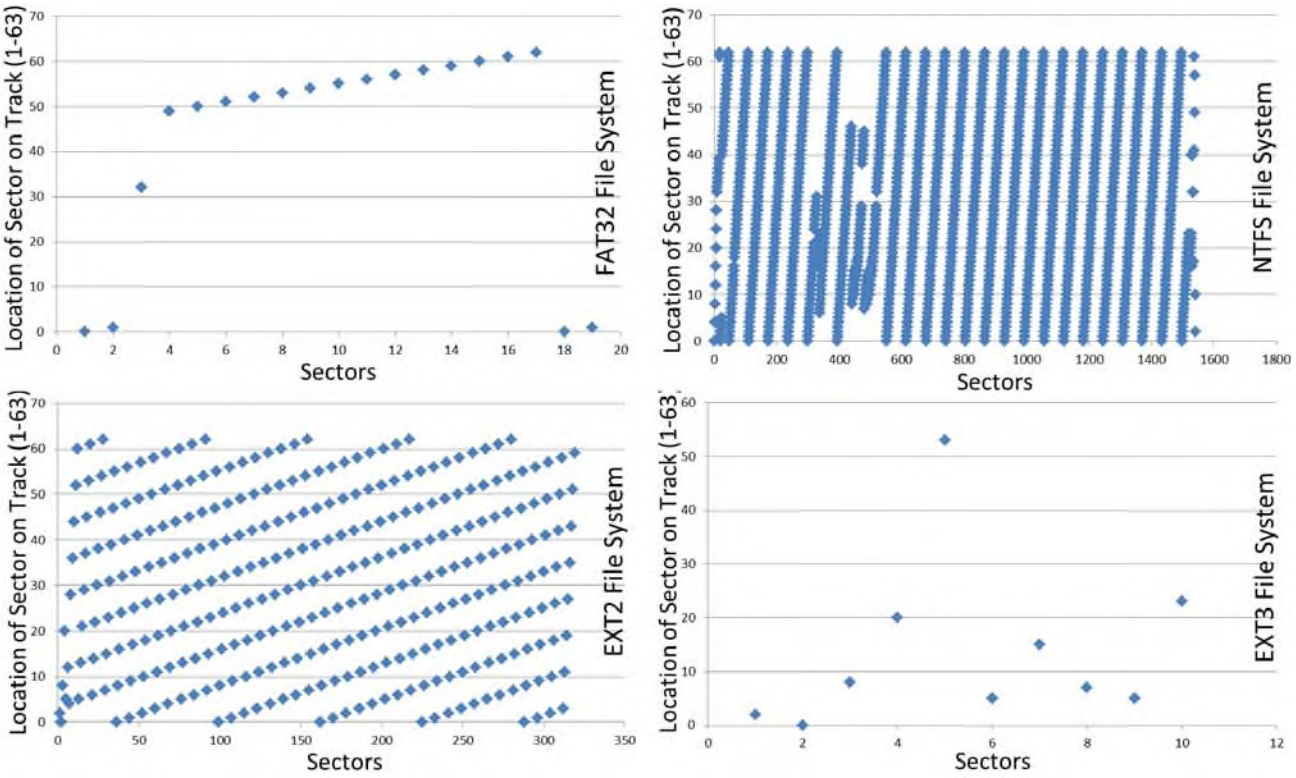


*Figure 1.1* *Read Trace for Test 1.*

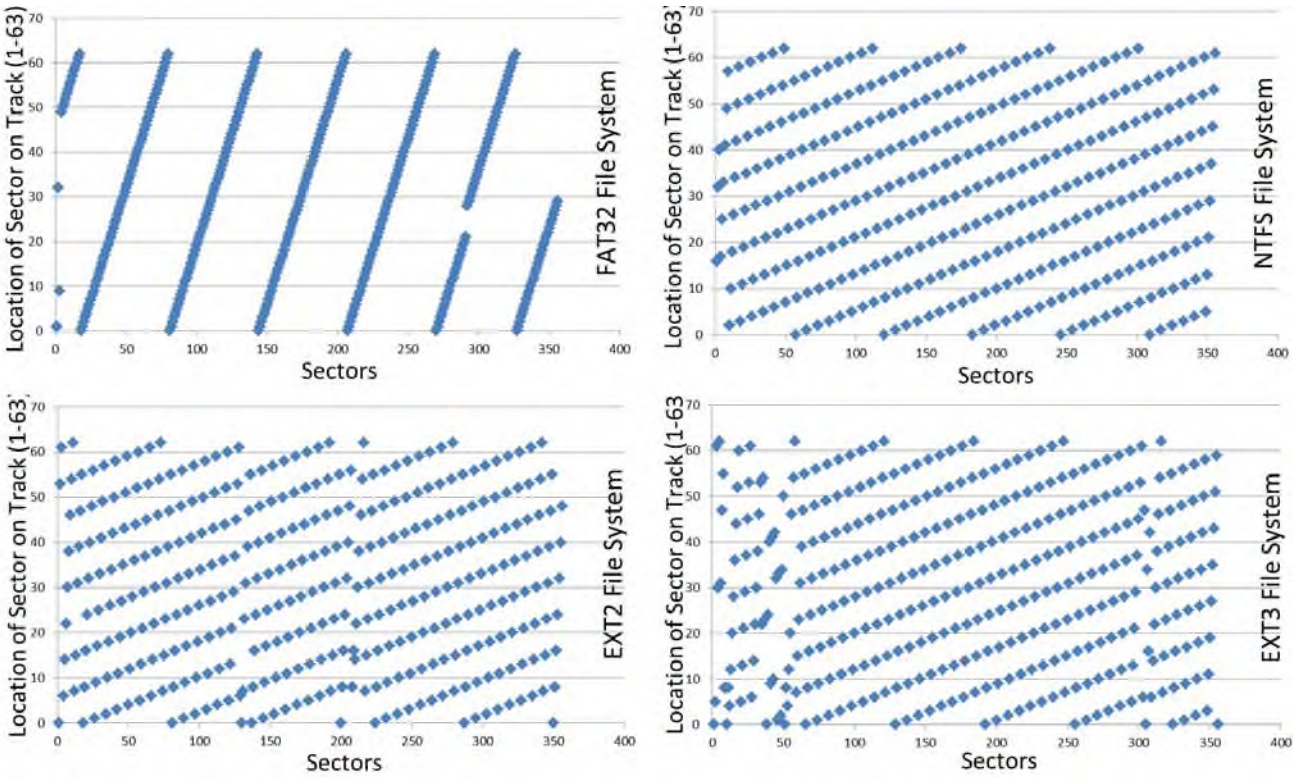


*Figure 1.2* *Write Trace for Test 1.*

*Figure 2.* *Sector Read Trace for Test 2.*

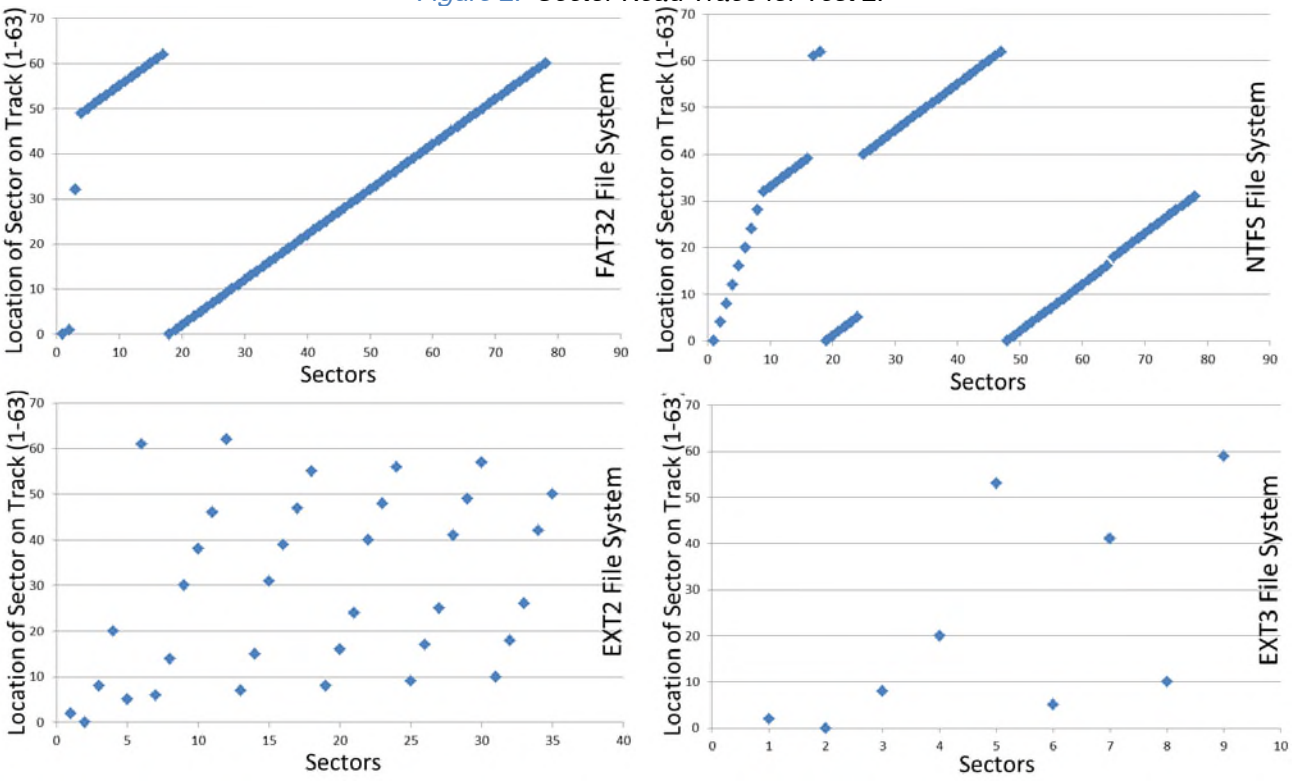


*Figure 5.* *Sector Read Trace for Test 5.*

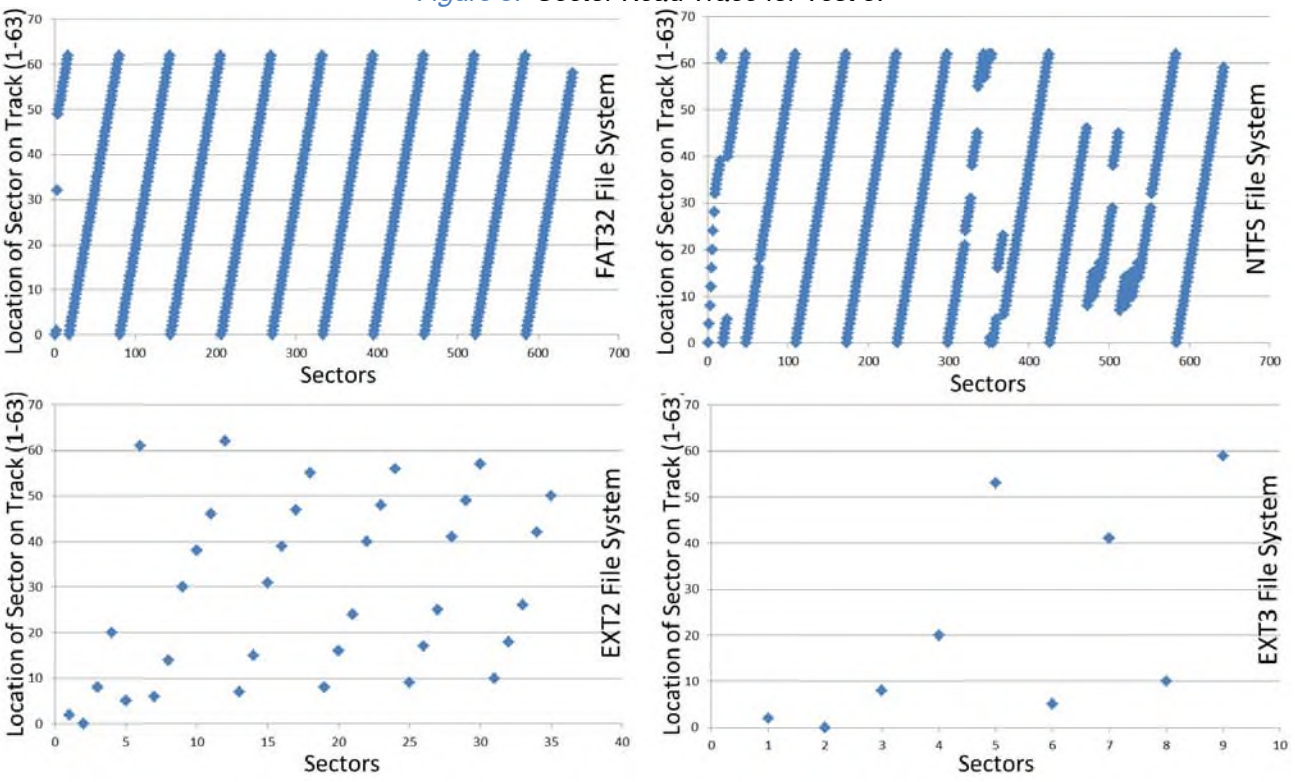


*Figure 3.* *Sector Read/Write Trace for Test 3.*

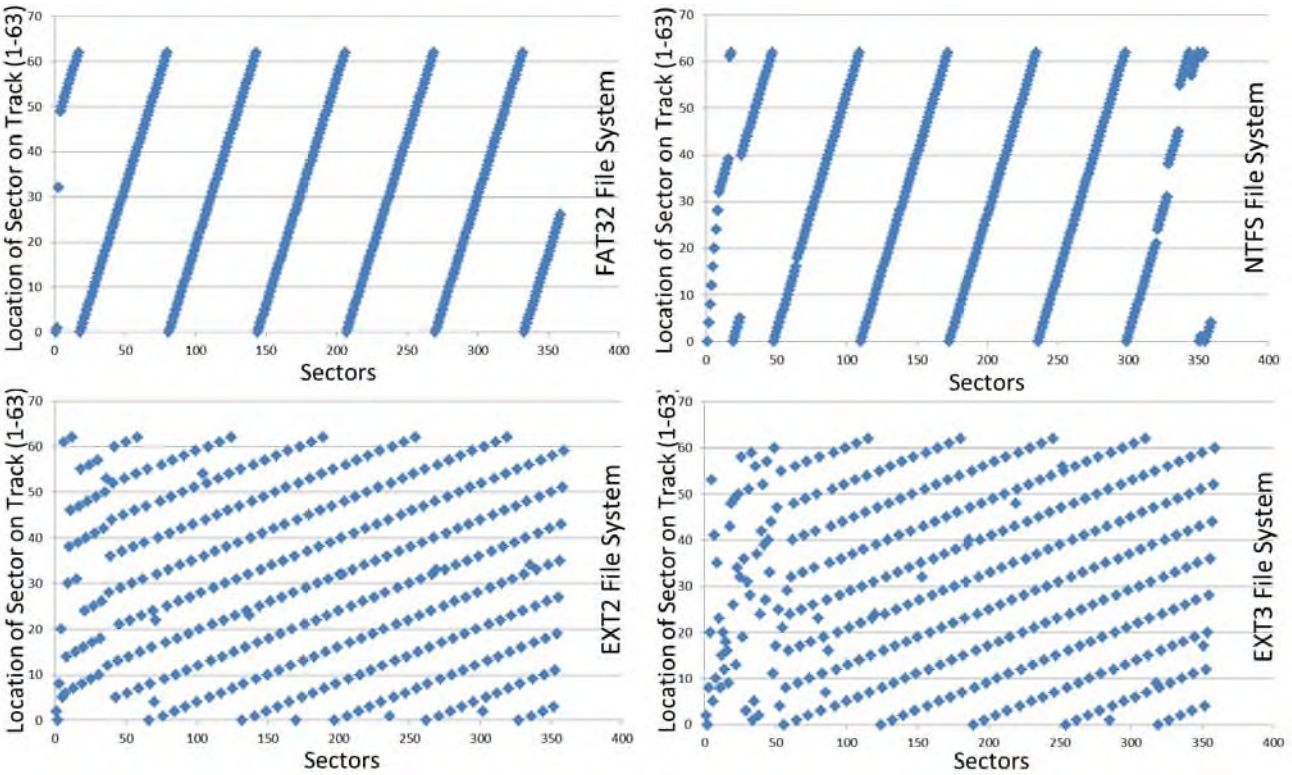


*Figure 3.1* *Read Trace for Test 3.*

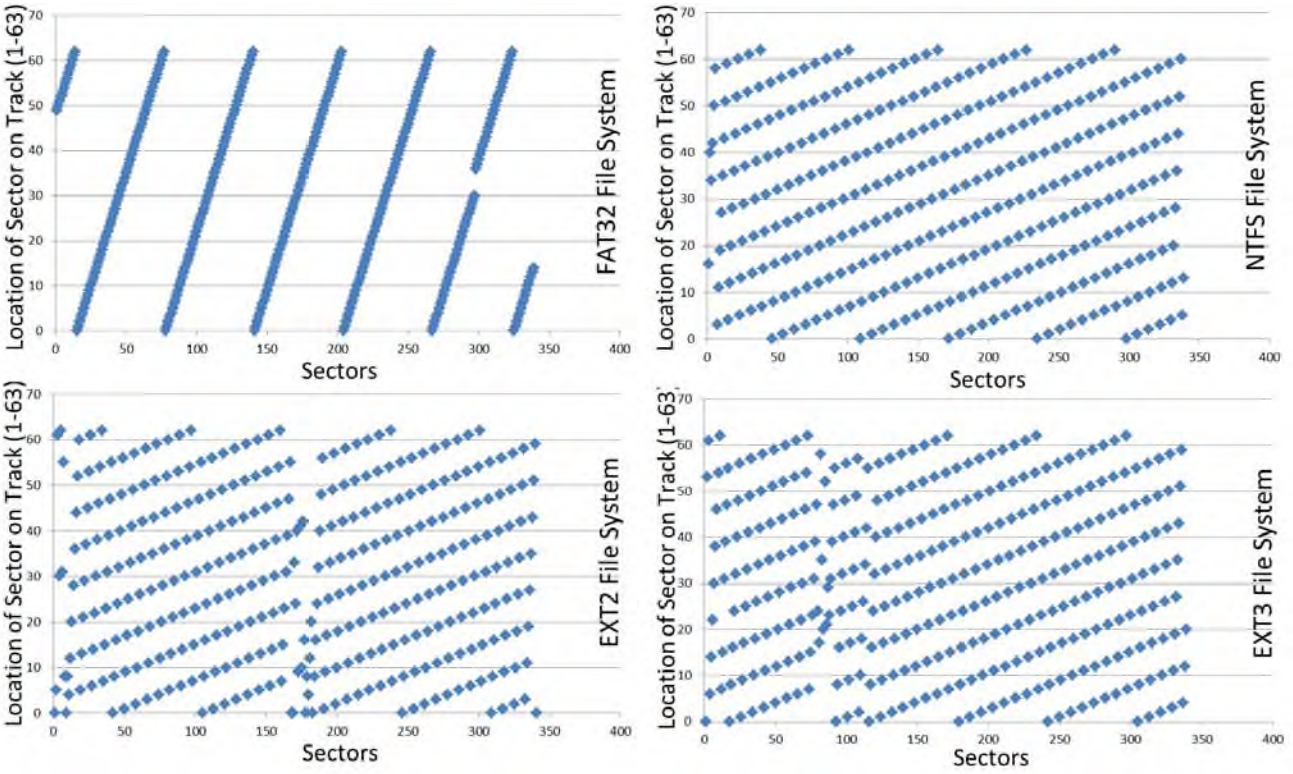


*Figure 3.2* *Write Trace for Test 3.*

*Figure 4.* ***Sector Read/Write Trace for Test 4.***

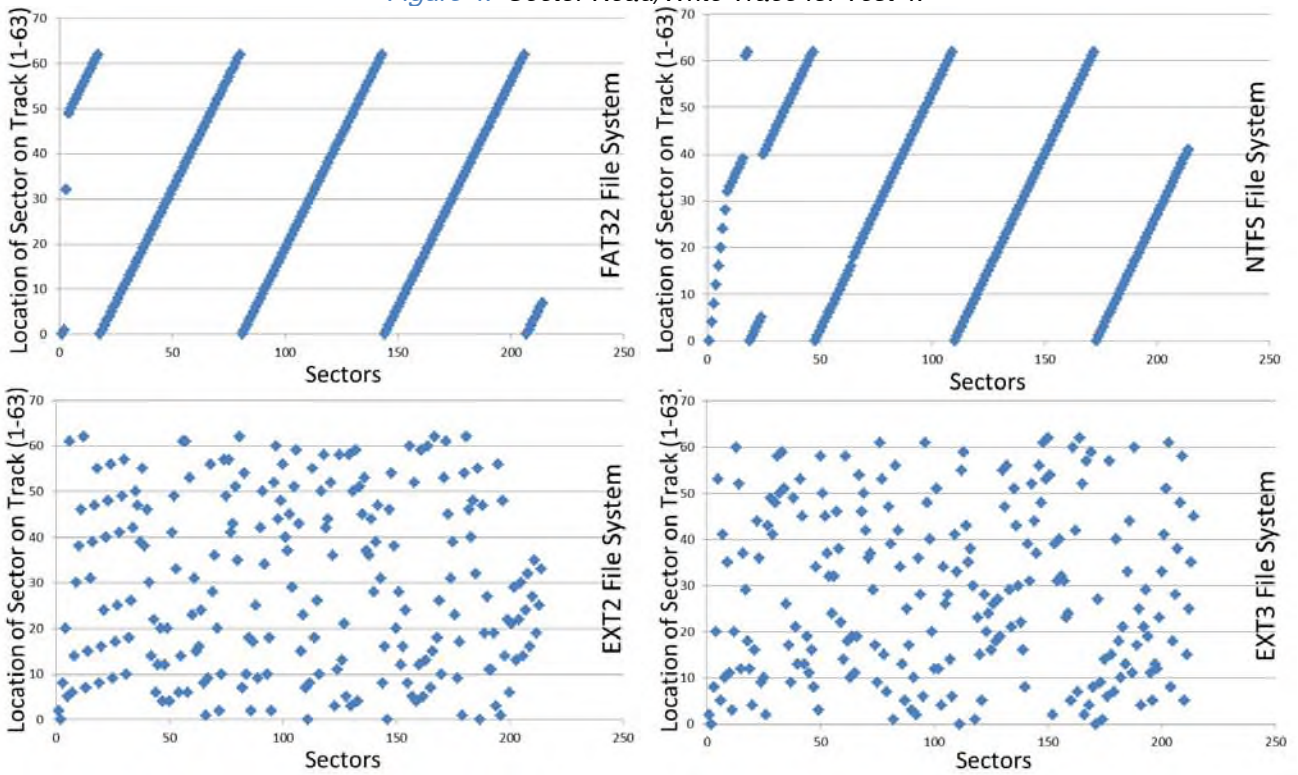


*Figure 4.1* ***Read Trace for Test 4.***

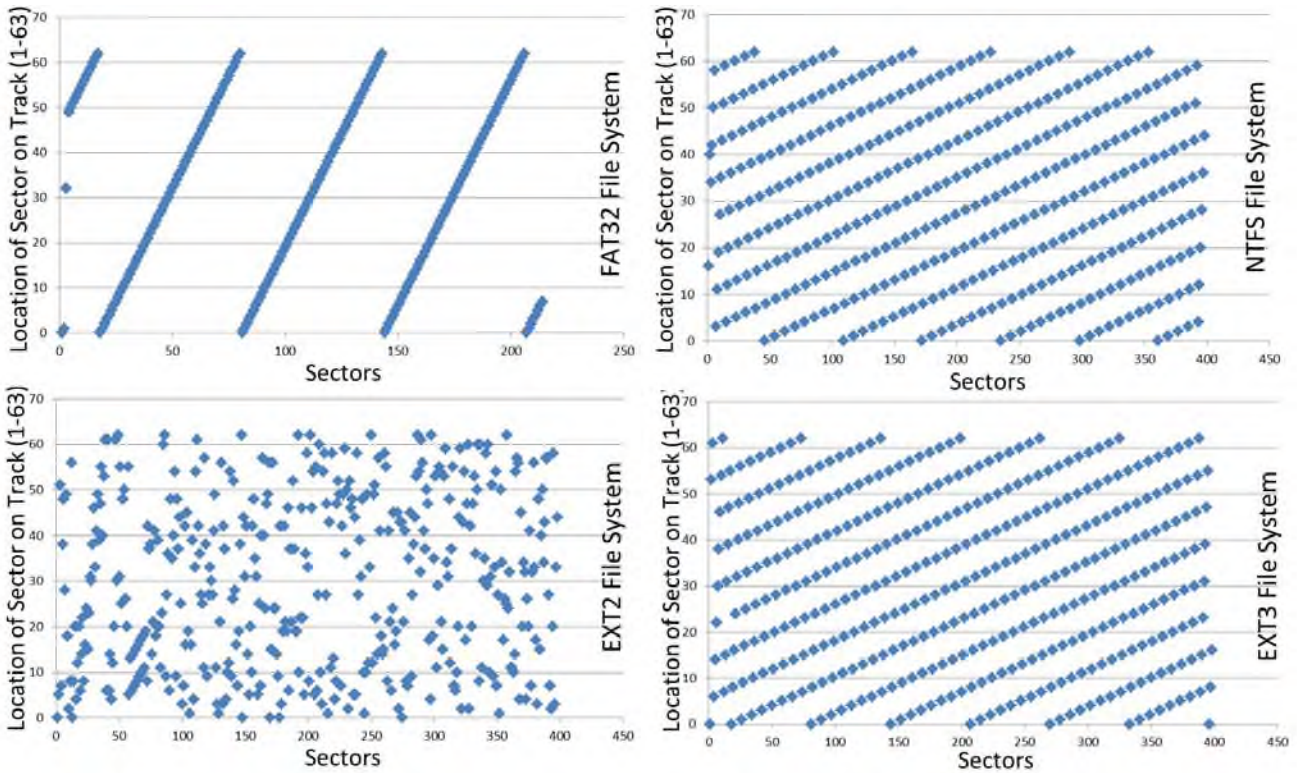


*Figure 4.2* ***Write Trace for Test 4.***

*Figure 6.* *Sector Read/Write Trace for Test 6.*

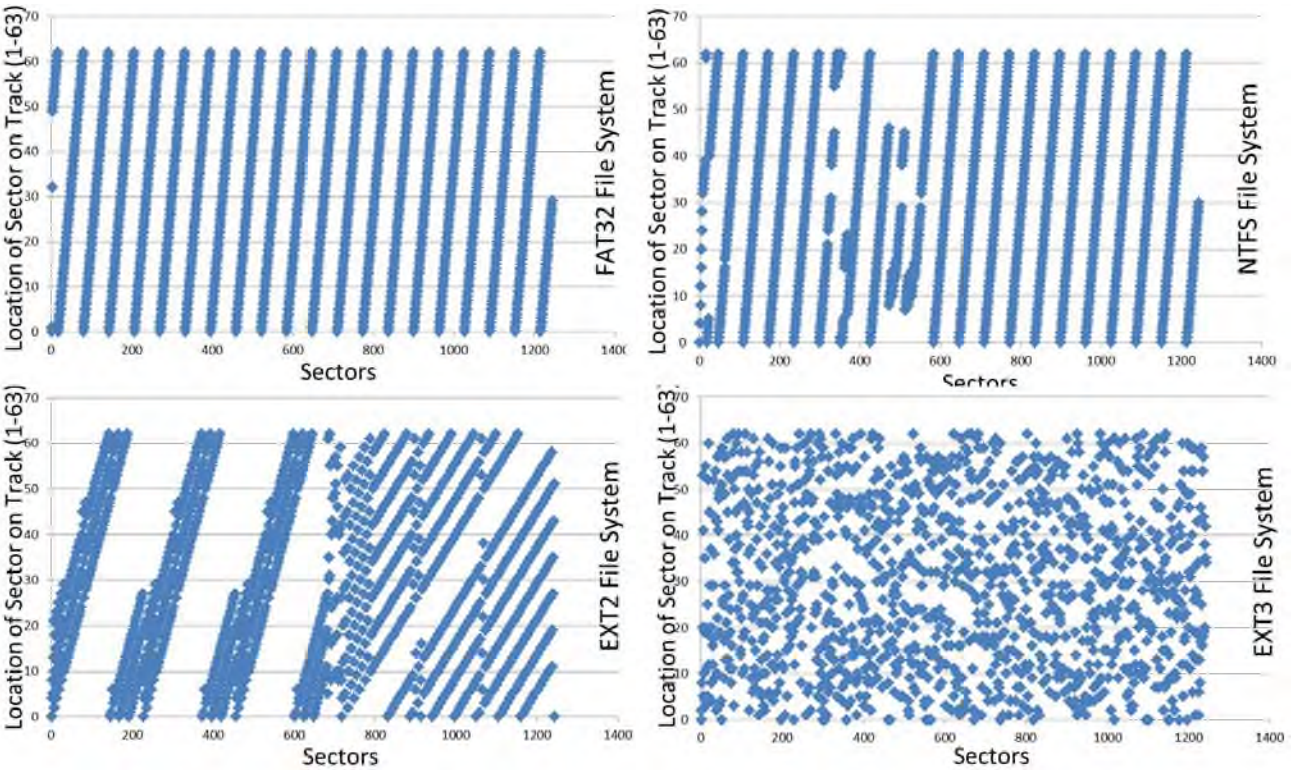


*Figure 6.1* *Read Trace for Test 6.*

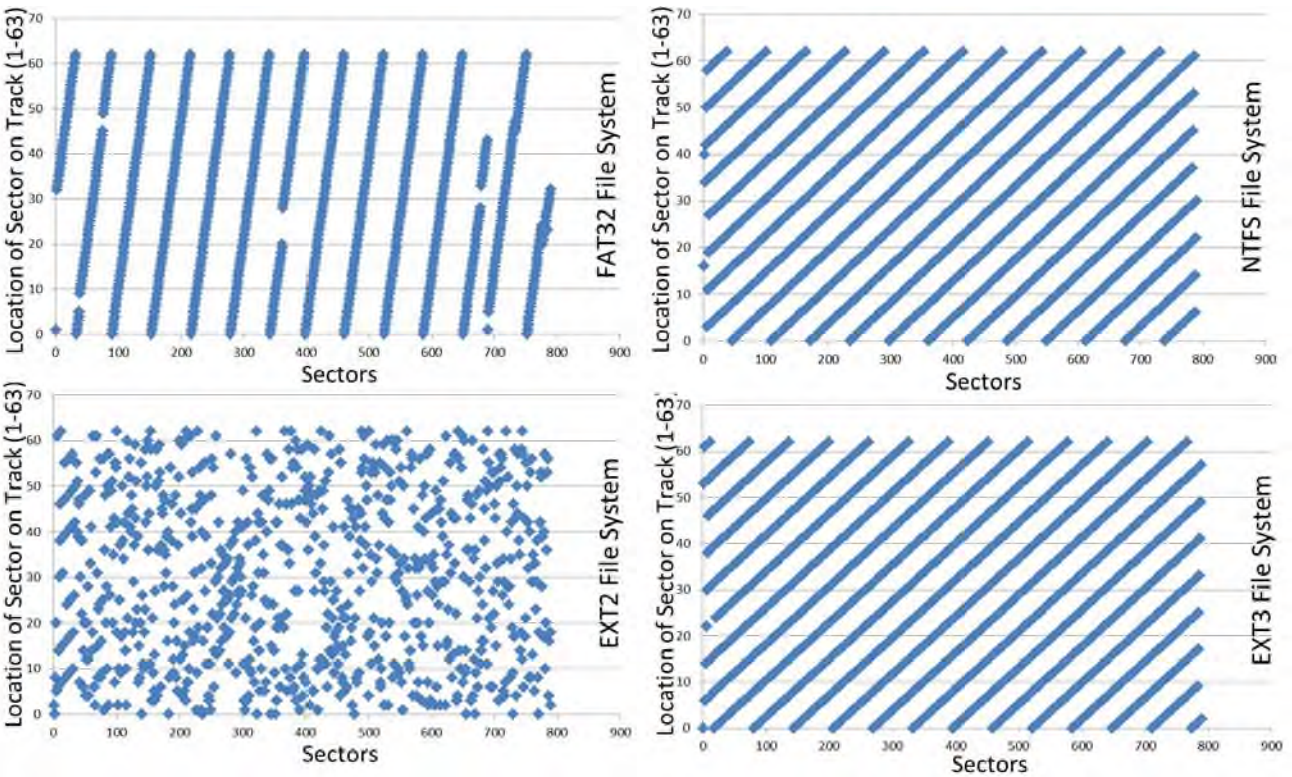


*Figure 6.2* *Write Trace for Test 6.*