File system usage in Windows NT 4.0

Werner Vogels
Department of Computer Science, Cornell University
vogels@cs.cornell.edu

Abstract

We have performed a study of the usage of the Windows NT File System through long-term kernel tracing. Our goal was to provide a new data point with respect to the 1985 and 1991 trace-based File System studies, to investigate the usage details of the Windows NT file system architecture, and to study the overall statistical behavior of the usage data.

In this paper we report on these issues through a detailed comparison with the older traces, through details on the operational characteristics and through a usage analysis of the file system and cache manager. Next to architectural insights we provide evidence for the pervasive presence of heavy-tail distribution characteristics in all aspect of file system usage. Extreme variances are found in session inter-arrival time, session holding times, read/write frequencies, read/write buffer sizes, etc., which is of importance to system engineering, tuning and benchmarking.

Categories and subject descriptors: C.4 [**Computer Systems Organization**]: performance of systems - *design studies*, D.4.3 [**Software**]: operating systems - *file systems management*.

1 Introduction

There is an extensive body of literature on usage patterns for file systems [1,5,9,11,14], and it has helped shape file system designs [8,13,17] that perform quite well. However, the world of computing has undergone major changes since the last usage study was performed in 1991; not only have computing and network capabilities increased beyond expectations, but the integration of computing in all aspects of professional life has produced new generations of systems and applications that no longer resemble the computer operations of the late eighties. These changes in the way computers are used may very well have an important impact on the usage of computer file systems.

SOSP-17 12/1999 Kiawah Island, SC

One of the changes in systems has been the introduction of a new commercial operating system, Microsoft's Windows NT, which has acquired an important portion of the professional OS market. Windows NT is different enough from Unix that Unix file systems studies are probably not appropriate for use in designing or optimizing Windows NT file systems.

These two observations have lead us to believe that new data about file systems usage is required, and that it would be particularly interesting to perform the investigation on a Windows NT platform.

In this paper we report on a file system usage study performed mainly during 1998 on the Windows NT 4.0 operating system. We had four goals for this study:

1. Provide a new data point with respect to earlier file system usage studies, performed on the BSD and Sprite operating systems.
2. Study in detail the usage of the various components of the Windows NT I/O subsystem, and examine undocumented usage such as the *FastIO* path.
3. Investigate the complexity of Windows NT file system interactions, with a focus on those operations that are not directly related to the data path.
4. Study the overall distribution of the usage data. Previous studies already hinted at problems with modeling outliers in the distribution, but we believe that this problem is more structural and warrants a more detailed analysis.

Next to these immediate goals, we wanted the investigation to result in a data collection that would be available for public inspection, and that could be used as input for file system simulation studies and as configuration information for realistic file system benchmarks.

The complexity of Windows NT file usage is easily demonstrated. When we type a few characters in the notepad text editor, saving this to a file will trigger 26 system calls, including 3 failed open attempts, 1 file overwrite and 4 additional file open and close sequences.

The rest of this paper is structured as follows: in section 2 we describe the systems we measured, and in section 3 and 4, we describe the way we collected the data and processed it. In section 5 we examine the file system layout information, and in section 6 we compare our tracing results with the BSD and Sprite traces. Section 7 contains a detailed analysis of the distribution aspects of our collected data. Sections 8, 9 and 10 contain details about the operation of various Windows NT file system components.

In comparison with the Sprite and BSD traces	*Operational characteristics*
– Per user throughput remains low, but is about 3 times higher (24 Kbytes/sec) than in Sprite (8 Kb/sec) – Files opened for data access are open for increasingly shorter periods: 75% of files remain open for less then 10 milliseconds versus a 75^{th} percentile of 250 milliseconds in Sprite. – Most accessed files are short in length (80% are smaller than 26 Kbytes), which is similar to Sprite. – Most access (60%) to files is sequential, but there is a clear shift towards random access when compared to Sprite. – The size of large files has increased by an order of magnitude (20% are 4Mbytes or larger), and access to these files accounts for the majority of the transferred bytes. – 81% of new files are overwritten within 4 milliseconds (26%) or deleted within 5 seconds (55%).	– The burstiness of the file operations has increased to the point where it disturbs the proper analysis of the data. – Control operations dominate the file system requests: 74% of the file opens are to perform a control or directory operation. – In 60% of the file read requests the data comes from the file cache. – In 92% of the open-for-read cases a single prefetch was sufficient to load the data to satisfy all subsequent reads from the cache. – The FastIO path is used in 59% of the read and 96% of the write requests. – Windows NT access attributes such as *temporary file*, *cache write- through*, *sequential access only*, can improve access performance significantly but are underutilized.
Trace data distribution characteristics	*File system content*
– There is strong evidence of extreme variance in all of the traced usage characteristics. – All the distributions show a significant presence of heavy-tails, with values for the Hill estimator between 1.2 and 1.7, which is evidence of infinite variance. – Using Poisson processes and Normal distributions to model file system usage will lead to incorrect results.	– Executables, dynamic loadable libraries and fonts dominate the file size distribution. – 94% of file system content changes are in the tree of user *profiles* (personalized file cache). – Up to 90% of changes in the user's profile occur in the WWW cache. – The time attributes recorded with files are unreliable

Table 1: Summary of observations

Section 11 touches on related work and section 12 summarizes the major points of the study. An overview of our observations can be found in table 1.

2 Systems under study

We studied a production environment in which five distinct categories of usage are found:

Walk-up usage. Users make use of a pool of available systems located in a central facility. The activities of these users vary from scientific analysis and program development to document preparation.

Pool usage. Groups of users share a set of dedicated systems, located near their work places. These users mainly are active in program development, but also perform a fair share of multimedia, simulation and data processing.

Personal usage. A system is dedicated to a single user and located in her office. The majority of the activities is in the category of collaborative style applications, such as email and document preparation. A smaller set of users uses the systems for program development.

Administrative usage. All these systems are used for a small set of general support tasks: database interaction, collaborative applications, and some dedicated administrative tools.

Scientific usage. These systems support major computational tasks, such as simulation, graphics processing, and statistical processing. The systems are dedicated to the small set of special applications.

The first four categories are all supported by Pentium Pro or Pentium II systems with 64-128 Mb memory and a 2-6 GB local IDE disk. The *pool* usage machines are in general more powerful (300-450 MHz, some dual processors), while the other machines are all in the 200 MHz range. The *scientific* usage category consists of Pentium II 450 Xeon dual and quad processors with a minimum of 256 MB of memory and local 9-18 GB SCSI Ultra-2 disks. All systems ran Windows NT 4.0 with the latest service packs applied. At the time of the traces the age of file systems ranged from 2 months to 3 years, with an average of 1.2 years.

There is central network file server support for all users. Only a limited set of personal workstations is supported through a backup mechanism, so central file storage is implicitly encouraged. All systems are connected to the network file servers through a 100 Mbit/sec switched Ethernet. The users are organized in three different NT domains, one for the walk-up usage, one general usage and one for experiments. The experimental domain has a trust relationship with the general domain and network file services are shared. The walk-up domain is separated from the other domains through a network firewall and has its own network file services.

From the 250 systems that were available for instrumentation, we selected a set of 45 systems based on privacy concerns and administrative accessibility. A subset of these systems was traced for 3 periods of 2 weeks during the first half of 1998 while we adjusted the exact type and amount of data collected. Some of the changes were related to the fact that our study was of an exploratory nature and the data collection had to be adjusted based on the initial results of the analysis. Other adjustments were related to our quest to keep the amount of data per trace record to an absolute minimum, while still logging sufficient information to support the analysis. We were not always successful as, for example, logging only the read request size is of limited use if the bytes actually read are not also logged. The analysis reported in this paper is based on a final data collection that ran for 4 weeks in November and December of 1998. The 45 systems generated close to 19 GB of trace data over this period.

Since then we have run additional traces on selected systems to understand particular issues that were unclear in the original traces, such as burst behavior of paging I/O, reads from compressed large files and the throughput of directory operations.

3 Collecting the data

The systems were instrumented to report two types of data: 1) snapshots of the state of the local file systems and 2) all I/O requests sent to the local and remote file systems. The first type is used to provide basic information about the initial state of the file system at the start of each tracing period and to establish the base set of files toward which the later requests are directed. In the second type of data all file system actions are recorded in real-time.

On each system a trace agent is installed that provides an access point for remote control of the tracing process. The trace agent is responsible for taking the periodic snapshots and for directing the stream of trace events towards the collection servers. The collection servers are three dedicated file servers that take the incoming event streams and store them in compressed formats for later retrieval. The trace agent is automatically started at boot time and tries to connect to a collection server; if it succeeds, it will initiate the local data collection. If a trace agent loses contact with the collection servers it will suspend the local operation until the connection is re-established.

3.1 File system snapshots

Each morning at 4 o'clock a thread is started by the trace agent server to take a snapshot of the local file systems. It builds this snapshot by recursively traversing the file system trees, producing a sequence of records containing the attributes of each file and directory in such a way that the original tree can be recovered from the sequence. The attributes stored in a *walk* record are the file name and size, and the creation, last modify and last access times. For directories the name, number of files entries and number of subdirectories is stored. Names are stored in a short form as we are mainly interested in the file type, not in the individual names. On FAT file systems the creation and last access times are not maintained and thus ignored.

The trace agent transfers these records to the trace collection server, where they are stored in a compressed format. Access to the collection files is through an OLE/DB provider, which presents each file as two database tables: one containing the directory and the other containing file information.

Producing a snapshot of a 2 GB disk takes between 30 and 90 seconds on a 200 MHz P6.

3.2 File system trace instrumentation

To trace file system activity, the operating system was instrumented so that it would record all file access operations. An important subset of the Windows NT file system operations are triggered by the virtual memory manager, which handles executable image loading and file cache misses through its memory mapped file interface. As such, it is not sufficient to trace at the system call level as was done in earlier traces. Our trace mechanism exploits the Windows NT support for transparent layering of device drivers, by introducing a filter driver that records all requests sent to the drivers that implement file systems. The trace driver is attached to each driver instance of a local file system (excluding removable devices), and to the driver that implements the network redirector, which provides access to remote file systems through the CIFS protocol.

All file systems requests are sent to the I/O manager component of the Windows NT operating system, regardless of whether the request originates in a user-level process or in another kernel component, such as the virtual memory manager or the network file server. After validating the request, the I/O manager presents it to the top-most device-driver in the driver chain that handles the volume on which the file resides. There are two driver access mechanisms: one is a generic packet based request mechanism, in which the I/O manager sends a packet (an *IRP* -- I/O request packet) describing the request, to the drivers in the chain sequentially. After handling a request packet a driver returns it to the I/O manager, which will then send it to the next device. A driver interested in post-processing of the request, after the packet has been handled by its destination driver, modifies the packet to include the address of a callback routine. A second driver access mechanism, dubbed *FastIO*, presents a direct method invocation mechanism: the I/O manager invokes a method in the topmost driver, which in turn invokes the same method on the next driver, and so on. The FastIO path is examined in more detail in section 10.

The trace driver records 54 IRP and FastIO events, which represent all major I/O request operations. The specifics of each operation are stored in fixed size records in a memory buffer, which is periodically flushed to the collection server. The information recorded depends on the particular operation, but each record contains at least a reference to the file object, IRP, File and Header Flags, the requesting process, the current byte offset and file size, and the result status of the operation. Each record receives two timestamps: one at the start of the operation and the other at completion time. These time stamps have a 100 nanosecond granularity. Additional information recorded depends on the particular operation, such as offset, length and returned bytes for the read and write operations, or the options and attributes for the create operation. An additional trace record is written for each new file object, mapping object id to a file name.

The trace driver uses a triple-buffering scheme for the record storage, with each storage buffer able to hold up to 3,000 records. An idle system fills this size storage buffer in an hour; under heavy load, buffers fill in as little as 3-5 seconds. Were the buffers to fill in less than 1 second, the increased communication latency between the host and server could lead to the overflow of the tracing module storage buffer. The trace agent would detect such an error, but this never occurred during our tracing runs.

Kernel profiling has shown the impact of the tracing module to be acceptable; under heavy IRP load the tracing activity contributed up to 0.5% of the total load on a 200 MHz P6.

In a 24-hour period the file system trace module would record between 80 thousand and 1.4 million events.

3.3 Executable and paging I/O

Windows NT provides functionality for memory mapped files, which are used heavily by two system services: (1) the loading of executables and dynamic loadable libraries is based on memory mapped files, and (2) the cache manager establishes a file mapping for each file in its cache, and uses the page fault mechanism to trigger the VM manager into loading the actual data into the cache. This tight integration of file system, cache manager and virtual memory manager poses a number of problems if we want to accurately account for all the file system operations.

The VM manager uses IRPs to request the loading of data from a file into memory and these IRPs follow the same path as regular requests do. File systems can recognize requests from the VM through a *PagingIO* bit set in the packet header. When tracing file systems one can ignore a large portion of the paging requests, as they represent duplicate actions: a request arrives from process and triggers a page fault in the file cache, which triggers a paging request from the VM manager. However, if we do ignore paging requests we would miss all paging that is related to executable and dynamic loadable library (dll) loading, and other use of memory mapped files. We decided to record all paging requests and filter out the cache manager induced duplicates during the analysis process.

We decided in favor of this added complexity, even though it almost doubled the size of our traces, because of the need for accuracy in accounting for executable-based file system requests. In earlier traces the *exec* system call was traced to record the executable size, which was used to adjust the overall trace measurements. In Windows NT this is not appropriate because of the optimization behavior of the Virtual Memory manager: executable code pages frequently remain in memory after their application has finished executing to provide fast startup in case the application is executed again.

3.4 Missing and noise data

We believe the system is complete in recording all major file system IO events, which is sufficient for getting insight into general Windows NT file system usage. There are many minor operations for which we did not log detailed information (such as locking and security operations), but they were outside of the scope of our study. During our analysis we found one particular source of noise: the local file systems can be accessed over the network by other systems. We found this access to be minimal, as in general it was used to copy a few files or to share a test executable. Given the limited impact of these server operations we decided to not remove them from the trace sets.

4 The data analysis process

The data analysis presented us with a significant problem: the amount of data was overwhelming. The trace data collection run we are reporting on in this paper totaled close to 20 GB of data, representing over 190 million trace records. The static snapshots of the local disks resulted in 24 million records.

Most related tracing research focuses on finding answers to specific sets of questions and hypotheses, which could be satisfied through the use of extensive statistical techniques, reducing the analysis to a number crunching exercise. Given the exploratory nature of our study we needed mechanisms with which we could *browse* through the data and search for particular patterns, managing the exposed level of details. Developing a representation of the data such that these operations could be performed efficiently on many millions of records turned out to be a very hard problem.

We were able to find a solution by realizing that this was a problem identical to the problems for which there is support in data-warehousing and on-line analytical processing (OLAP). We developed a de-normalized star schema for the trace data and constructed corresponding database tables in SQL-server 7.0. We performed a series of summarization runs over the trace data to collect the information for the dimension tables. Dimension tables are used in the analysis process as the category axes for multidimensional cube representations of the trace information. Most dimensions support multiple levels of summarization, to allow a *drill-down* into the summarized data to explore various levels of detail. An example of categorization is that a mailbox file with a *.mbx* type is part of the *mail files* category, which is part of the *application files* category.

We departed from the classical data-warehouse model in that we used two fact tables (the tables that hold all the actual information), instead of one. The first table (*trace*) holds all the trace data records, with key references to dimension tables. The second table (*instance*) holds the information related to each FileObject instance, which is associated with a single file open-close sequence, combined with summary data for all operations on the object during its life-time. Although the second table could be produced by the OLAP system, our decision to use two fact tables reduced the amount of storage needed in the trace table by references to the instance table, reducing the processing overhead on operations that touch all trace records.

The use of a production quality database system provided us with a very efficient data storage facility. An operation that would touch all trace data records, such as calculation of the basic statistical descriptors (avg, stdev, min, max) of request inter-arrival times, runs at 30% of the time a hand optimized C-process on the original trace data takes. More complex statistical processing that could not be expressed in SQL or MDX was performed using the SPSS statistical processing package that directly interfaces with the database.

Because we processed the data using different category groupings (e.g. per day, per node, per user, per process, etc.) our analysis frequently did not result in single values for the statistical descriptors. In the text we show the ranges of these values or, where relevant, only the upper or lower bound.

5 File system content characteristics

To accurately analyze the real-time tracing results we needed to examine the characteristics of the set of files that were to be accessed in our trace sessions. For this we took snapshots of each of the file systems that was used for tracing as described in section 3.1. We collected file names and sizes, and creation and access times, as well as directory structure and sizes.

We supplemented this data with periodic snapshots of the user directories at the network file severs. However, the results of these snapshots cannot be seen as the correct state of the system from which the real-time tracing was performed, as they included the home directories of more users than those being traced. The network file server information was used to establish an intuitive notion of the differences between local and network file systems.

Recently Douceur and Bolosky have published a study on the content of over 10,000 Windows NT file systems within Microsoft [4]. Most of our findings are consistent with their conclusions, and we refer to their paper for a basic understanding of Windows NT file system content. Our content tracing allowed us to track the state of the file systems over time, and in this section we report from that specific view.

We see that the local file systems have between 24,000 and 45,000 files, that the file size distribution is similar for all systems, and that the directory depth and sizes are almost identical. File systems are between 54% and 87% full.

The network server file systems are organized into *shares*, which is a remote mountable sub-tree of a file system. In our setting each share represents a user's home directory. There was no uniformity in size or content of the user shares; sizes ranged from 500 Kbytes to 700 Mbytes and number of files from 150 to 27,000. The directory characteristics exhibit similar variances.

Decomposition of the local and network file systems by file type shows a high variance within the categories as well as between categories. What is remarkable is that this file type diversity does not appear to have any impact on the file size distribution; the large-sized outliers in the size distribution dominate the distribution characteristics. If we look again at the different file types and weigh each type by file size we see that the file type distribution is similar for all system types, even for the network file systems. The file size distribution is dominated by a select group of file-types that is present in all file systems. For local file systems the size distribution is dominated by executables, dynamic loadable libraries and fonts, while for the network file system the set of large files is augmented with development databases, archives and installation packages.

When we examine the local file systems in more detail we see that the differences in file system state are determined by two factors: 1) the file distribution within the user's *profile*, and 2) the application packages installed. Most of our test systems have a limited number of user specific files in the local file system, which are generally stored on the network file servers.

Of the user files that are stored locally between 87% and 99% can be found in the *profile* tree (\winnt\profiles\<username>). Each profile holds all the files that are unique to a user and which are stored by the system at a central location. These files are downloaded to each system the user logs into from a profile server, through the *winlogon* process. This includes files on the user's desktop, application specific data such as mail files, and the user's world-wide-web cache. At the end of each session the changes to the profiles are migrated back to the central server. When we detect major differences between the systems, they are concentrated in the tree under the \winnt\profiles directory. For the "*Temporary Internet Files*" WWW cache we found sizes between 5 and 45 Mbytes and with between 2,000 and 9,500 files in the cache.

		Windows NT	Sprite	BSD
10-minute intervals	Max number of active users	45	27	31
	Average number of active users	28.9 (21.6)	9.1 (5.1)	12.6
	Average throughput for a user in an interval	24.4 (57.9)	8.0 (36)	0.40 (0.4)
	Peak throughput for an active user	814	458	NA
	Peak throughput system wide	814	681	NA
10-second intervals	Max number of active users	45	12	NA
	Average number of active users	6.3 (15.3)	1.6 (1.5)	2.5 (1.5)
	Average throughput for a user in an interval	42.5 (191)	47.0 (268)	1.5 (808)
	Peak throughput for an active user	8910	9871	NA
	Peak throughput system wide	8910	9977	NA

Table 2. User activity. The throughput is reported in Kbytes/second (with the standard deviation in parentheses).

A second influence on the content characteristics of the file system is the set of application packages that are installed. Most general packages such as Microsoft Office or Adobe Photoshop have distribution dynamics that are identical to the Windows NT base system and thus have little impact on the overall distribution characteristics. Developer packages such as the Microsoft Platform SDK, which contains 14,000 files in 1300 directories, create a significant shift in file-type count and the average directory statistics.

When we examine the changes in the file systems over time, similar observations can be made. Major changes to a Windows NT file system appear when a new user logs onto a system, which triggers a profile download, or when a new application package is installed. Without such events, almost all of the measured changes were related to the current user's activities, as recorded in her profile. A commonly observed daily pattern is one where 300-500 files change or are added to the system, with peaks of up to 2,500 and 3,000 files, up to 93% of which are in the WWW cache.

Changes to user shares at the network file server occur at much slower pace. A common daily pattern is where 5-40 files change or are added to the share, with peaks occurring when the user installs an application package or retrieves a large set of files from an archive.

If we look at the age of files and access patterns over time, a first observation to make is that the three file times recorded with files (creation, last access, last change) are unreliable. These times are under application control, allowing for changes that cause inconsistencies. For example, in 2-4% of the examined cases, the last change access is more recent than the last access times. Installation programs frequently change the file creation time of newly installed files to the creation time of the file on the installation medium, resulting in files that have creation times of years ago on file systems that are only days or weeks old. In [18] the creation times were also not available and a measure used to examine usage over time was the *functional lifetime*, defined as the difference between the last change and the last access. We believe that these timestamps in Windows NT are more accurate than the creation time, as the file system is the main modifier of these timestamps, but we are still uncertain about their correctness, and as such we cannot report on them.

6 BSD & Sprite studies revisited

One of the goals of this study was to provide a new data point in relation to earlier studies of the file system usage in BSD 4.2 and the Sprite operating systems. These studies reported their results in three categories: 1) user activity (general usage of the file system on a per user basis), 2) access patterns (read/write, sequential/random), and 3) file lifetimes. A summary of the conclusions of this comparison can be found in table 1.

The Windows NT traces contain more detail than the BSD/Sprite traces, but in this section we will limit our reporting to the type of data available in the original studies.

Strong caution: when summarizing the trace data to produce tables identical to those of the older traces, we resort to techniques that are **not** statistically sound. As we will show in section 7, access rates, bytes transferred and most of the other properties investigated are not normally distributed and thus cannot be accurately described by a simple average of the data. We present the summary data in table 2 and 3 to provide a historical comparison.

6.1 User activity

Table 2 reports on the user activity during the data collection. The tracing period is divided into 10-minute and 10-second intervals, and the number of active users and the throughput per user is averaged across those intervals. In the BSD and Sprite traces it was assumed that 10 minutes was a sufficient period to represent a steady state, while the 10-second average would more accurately capture bursts of activity.

The earlier traces all reported on multi-user systems, while the Windows NT systems under study are all configured for a single user. A user and thus a system are considered to be active during an interval if there was any file system activity during that interval that could be attributed to the user. In Windows NT there is a certain amount of background file system activity, induced by

File Usage	Accesses (%)				Bytes (%)				Type of transfer	Accesses (%)				Bytes (%)			
	W	-	+	S	W	-	+	S		W	-	+	S	W	-	+	S
Read-only	79	21	97	88	59	21	99	80	Whole file	68	1	99	78	58	3	96	89
									Other sequential	20	0	62	19	11	0	72	5
									Random	12	0	99	3	31	0	97	7
Write-only	18	3	77	11	26	0	73	19	Whole file	78	5	99	67	70	1	99	69
									Other sequential	7	0	51	29	3	0	47	19
									Random	15	0	94	4	27	0	99	11
Read/Write	3	0	16	1	15	0	70	1	Whole file	22	0	90	0	5	0	76	0
									Other sequential	3	0	28	0	0	0	14	0
									Random	74	2	100	100	94	9	100	0

Table 3. Access patterns, the *W* column holds the mean for the Windows NT traces, the *S* column holds the values from the Sprite traces. The – and + columns indicate the range for the values. All numbers are reported in percentages.

systems services, that was used as the threshold for the user activity test.

The result of the comparison is in table 2. The average throughput per user has increased threefold since the 1991 Sprite measurements. A remarkable observation is that this increase can only be seen for the 10-minute periods, for the 10-second period there was no such increase and the peak measurements are even lower.

The Sprite researchers already noticed that the increase in throughput per user was not on the same scale as the increase of processor power per user. They attributed this to the move from a system with local disk to a diskless system with network file systems. In our traces we are able to differentiate between local and network access, and when summarizing it appears that there is indeed such a difference in throughput. However detailed analysis shows that the difference can be completely attributed to the location of executables and large system-files such as fonts, which are all placed on the local disk.

One of the reasons for the high peak load in Sprite was the presence of large files from a scientific simulation. Although the scientific usage category in our traces uses files that are of an order of magnitude larger (100-300 Mbytes), they do not produce the same high peak loads seen in Sprite. These applications read small portions of the files at a time, and in many cases do so through the use of memory-mapped files.

The peak load reported for Windows NT was for a development station, where in a short period a series of medium size files (5-8 Mb), containing precompiled header files, incremental linkage state and development support data, was read and written.

6.2 File access patterns

The BSD and Sprite (and also the VMS [15]) traces all concluded that most access to files is sequential. Summaries of our measurements, as found in table 3, support this conclusion for Windows NT file access, but there is also evidence of a shift towards more randomized access to files when compared to the Sprite results.

A sequential access is divided into two classes: complete file access and partial file access. In the latter case all read and write accesses are sequential but the access does not start at the beginning of the file or transfers fewer bytes than the size of the file at close time.

The Windows NT traces do not support the trend seen

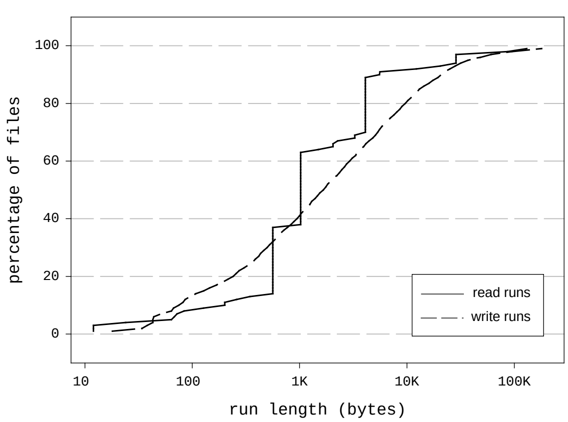

Figure 1. The cumulative distribution of the sequential run length of weighted by the number of files

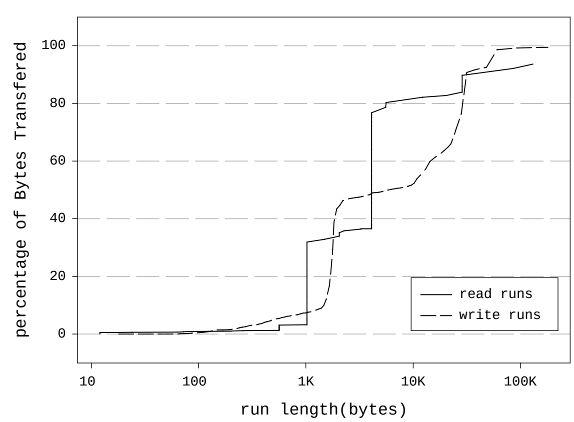

Figure 2. The cumulative distribution of the sequential run length of weighted by bytes transferred

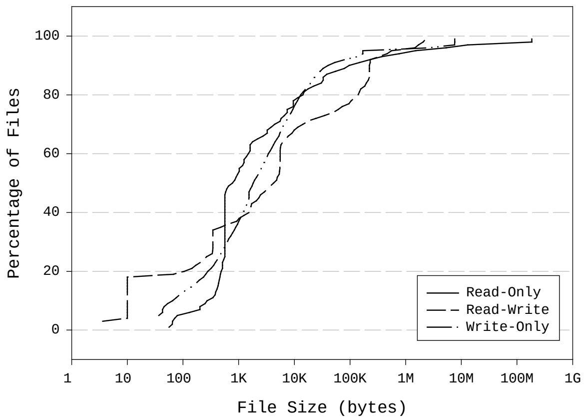

Figure 3. File size cumulative distribution, weighted by the number of files opened

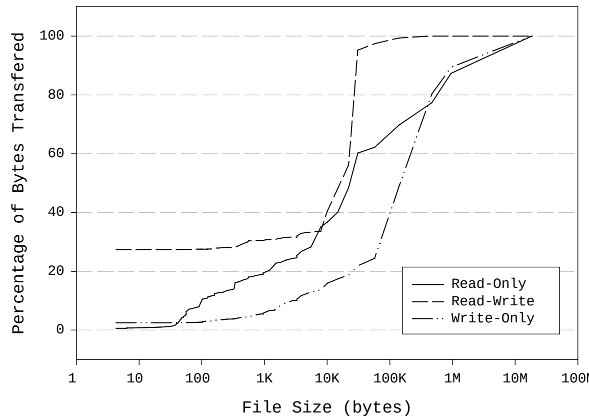

Figure 4. File size cumulative distribution, weighted by the number of bytes transferred

in Sprite, where there was a 10% increase in sequential access. On average 68% of the read-only accesses were whole-file sequential, versus 78% in the Sprite traces. A significant difference from Sprite is the amount of data transferred sequentially: in Sprite 89% of the read-only data was transferred sequentially versus 58% in the Windows NT traces. When comparing this type of trace summary there is a stronger presence of random access to data both in number of accesses and in the amount of data accessed for all file usage categories.

Another important access pattern examined is that of the *sequential runs*, which is when a portion of a file is read or written in a sequential manner. The prediction of a series of sequential accesses is important for effective caching strategies. When examining these runs we see that they remain short; the 80% mark for Sprite was below the 10 Kbytes, while in our traces we see a slight increase in run length with the 80% mark at 11 Kbytes (figure 1).

An observation about the Sprite traces was that most bytes were transferred in the longer sequential runs. The Windows NT traces support this observation, although the correlation is less prominent (figure 2). Access to large files shows increasing random access patterns, causing 15%-35% (in some traces up to 70%) of the bytes to be transferred in non-sequential manner.

If we look at all file open sessions for which data transfers where logged, not just those with sequential runs, we see that the 80% mark for the number of accesses changes to 24 Kbytes. 10% of the total transferred bytes were transferred in sessions that accessed at least 120 Kbytes.

When examining the size of files in relation to the number of sequential IO operations posted to them we see a similar pattern: most operations are to short files (40% to files shorter than 2K) while most bytes are transferred to large files (figures 3 and 4).

In Sprite the researchers found that, when examining the top 20% of file sizes, an increase of an order of magnitude was seen with respect to the BSD traces. This trend has continued: in the Windows NT traces the top 20% of files are larger than 4 Mbytes. An important contribution to this trend comes from the executables and dynamic loadable libraries in the distribution, which account for the majority of large files.

The last access pattern for which we examine the traces concerns the period of time during which a file is open. In this section we only look at file open sessions that have data transfer associated with them; sessions specific for control operation are examined in section 8. The results are presented in figure 5; about 75% of the files are open less than 10 milliseconds. This is a significant change when compared to the Sprite and BSD traces, which respectively measured a quarter-second and a half-second at 75%. The less significant difference between the two older traces was attributed to the fact that in the BSD traces the I/O was to

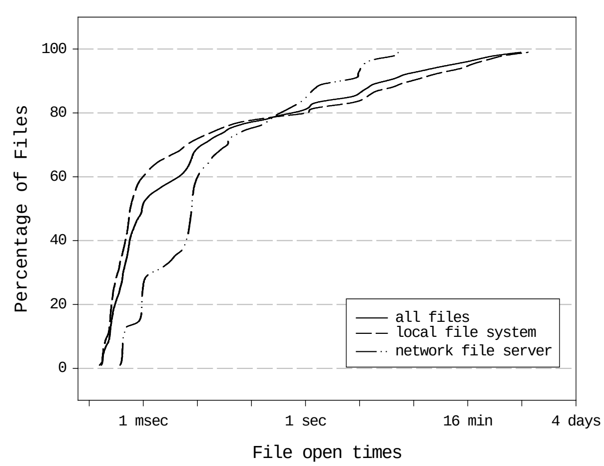

Figure 5. File open time cumulative distribution weighted by the number of files.

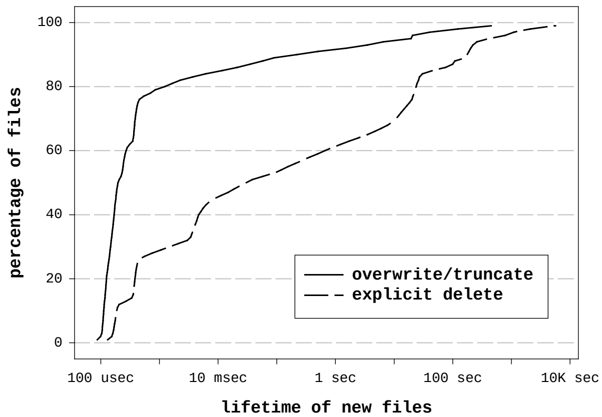

Figure 6. The lifetime of newly created files grouped by deletion method

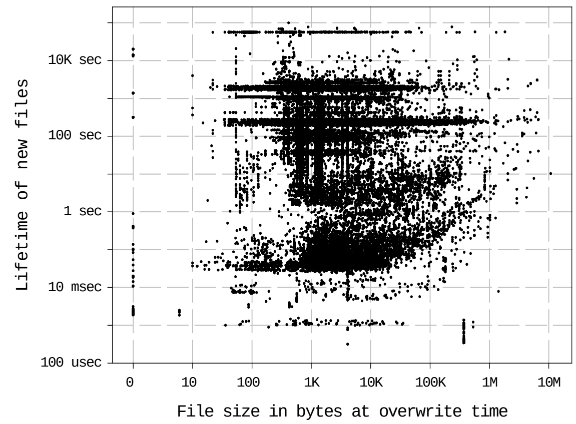

Figure 7. When examining file sizes at overwrite time, we cannot find a correlation between size and lifetime

local storage while in the Sprite the storage was accessed over the network. In the Windows NT traces we are able to examine these access times separately, and we found no significant difference in the access times between local and remote storage.

6.3 File lifetimes

The third measurement category presented in the BSD & Sprite traces is that of the lifetime of newly created files. The Sprite traces showed that between 65% and 80% of the new files were deleted within 30 seconds after they were created. In the Windows NT traces we see that the presence of this behavior is even stronger; up to 80% of the newly created files are deleted within 4 seconds of their creation.

In Windows NT we consider three sources for deletion of new files: (1) an existing file is truncated on open by use of a special option (37% of the delete cases), (2) a file is newly created or truncated and deleted using a delete control operation (62%), and (3) a file is opened with the *temporary file* attribute (1%).

In about 75% of the delete-through-truncate cases a file was overwritten within 4 milliseconds after it was created. The distribution shows a strong heavy tail with the top 10% having a lifetime of at least 1 minute, and up to 18 hours. If we inspect the time between the closing of a file and the subsequent overwrite action, we see that over 75% of these files are overwritten within 0.7 millisecond of the close.

In the case of explicitly deleted files, we see a higher latency between create and delete action. 72% of these files are deleted within 4 seconds after they were created and 60% 1.5 seconds after they were closed (see figure 6).

One of the possible factors in the difference in latency is related to which process deletes the file. In 94% of the overwrite cases, the process that overwrites the file also created it in the first place, while in 36% of the DeleteFile cases the same process deletes the file. A second factor is that there are no other actions posted to overwritten files, while in 18% of the DeleteFile cases, the file is opened one or more times between creation and deletion.

The *temporary file* attribute not only causes the file to be deleted at close time, but also prevents the cache manager's lazy writer threads from marking the pages containing the file data for writing to disk. Although it is impossible to extract the exact persistency requirements for temporary file usage from the traces, analysis suggests that at least 25%-35% of all the deleted new files could have benefited from the use of this attribute.

In 23% of the cases where a file was overwritten, unwritten pages were still present in the file cache when the overwrite request arrived. In the case of the CreateFile/DeleteFile sequence 5% of the newly created files were still had unwritten data present in the cache when deleted. Anomalous behavior was seen in 3% of the cases where the file was flushed from the cache by the application before it was deleted.

The apparent correlation between the file size and lifetime, as noticed by the Sprite researchers, is tremendously skewed by the presence of large files. In the Windows NT case only 4% of the deleted files are over 40 Kbytes and 65% of the files are smaller than 100 bytes. In the traces we could not find any proof that large temporary files have a longer lifetime. Figure 7 shows a plot of lifetime versus size of a trace sample, and although there are no large files in this plot that are deleted in less then 1 second, there is no statistical justification for a correlation between size and lifetime of temporary files.

7 Data distribution

When analyzing the user activity in section 6.1 we were tempted to conclude that for Windows NT the average throughput in general has increased, but that the average in burst load has been reduced. The use of simple averaging techniques allows us to draw such conclusions, in similar fashion one could conclude from the file access patterns

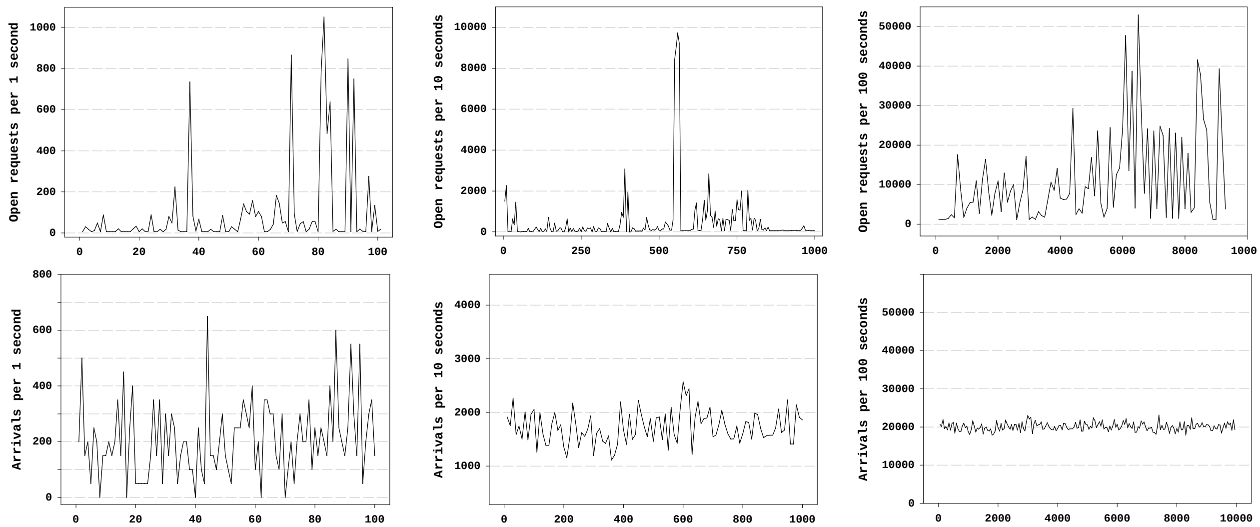

Figure 8. The top row displays the inter-arrival distribution of file open events at 3 different orders of magnitude. The bottom row contains a synthesized sample of a Poisson process with parameters estimated from the sample.

that most file accesses still occur in a read-only, whole-file sequential manner. If we examine the result of analysis of the file access in table 3 once more, the truly important numbers in that table are the ranges of values that were found for each of the statistical descriptives. The -/+ columns in the table represent the min/max values found when analyzing the traces of each trace separately.

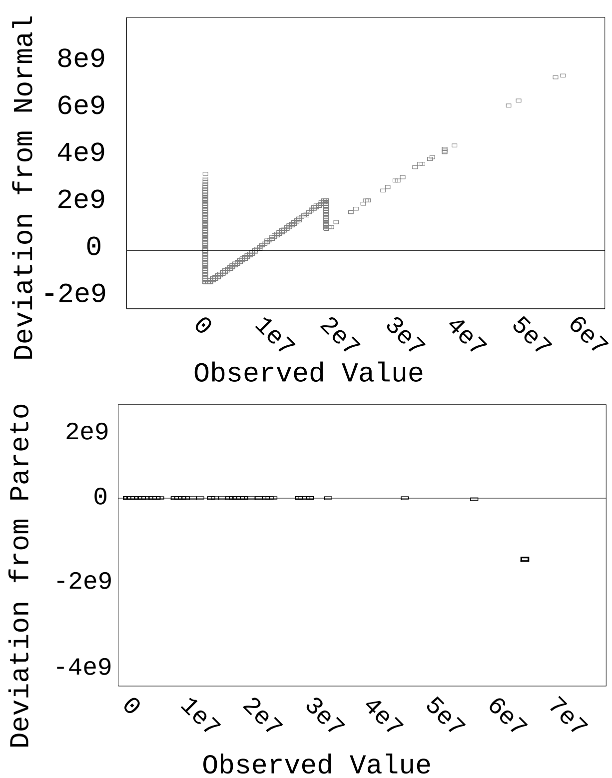

Figure 9. The arrival data from the sample used in figure 8 is tested against a Normal and a Pareto distribution through a QQ plot. The plot tests to what extend sample data follows a given distribution with estimated parameters.

When we have a closer look at the trace data and the statistical analysis of it, we find a significant variance in almost all variables that we can test. A common approach to statistically control burstiness, which is often the cause of the extreme variances, is to examine the data on various time scales. For example, in the previous two file system trace reports, the data was summarized over 10-second and 10-minute intervals, with the assumption that the 10-minute interval would smoothen any variances found in the traces.

If, for example, we consider the arrival rate of file system requests to be drawn from a Poisson distribution, we should see that the variances should diminish when we view the distribution at coarser time granularity. In figure 8 we compare the request arrival rates in one of our trace files, randomly chosen, with a synthesized sample from a Poisson distribution for which we estimated its mean and variance from the trace information (the details of this test are presented in [21]). When we view the samples at time scales with different orders of magnitude, we see that at larger time scales the Poisson sample becomes smooth, while the arrival data in our sample distribution continues to exhibit the variant behavior.

In almost all earlier file system trace research there is some notion of the impact of large files on the statistical analysis. In the Sprite research, for example, an attempt was made to discard the impact of certain categories of large files, by removing kernel development files from the traces. The result, however, did not remove the impact of large files, leading the researchers to conclude that the presence of large files was not accidental.

Analyzing our traces for the impact of outliers we find they are present in all categories. For example if we take the distribution of bytes read per open-close session, we see that the mean of the distribution is forced beyond the 90th percentile by the impact of large file read sessions. If we visually examine how the sample distribution from figure 8 departs from normality through a QQ plot (figure 9) we see that values in the quartiles support the evidence that the distribution is not normal. If we use a QQ plot to test the sample against a Pareto distribution, which is the simplest distribution that can be used to model heavy-tail behavior, we see an almost perfect match.

To examine the tail in our sample distribution we produced a log-log complementary distribution plot (figure 10). The linear appearance of the plot is evidence of the power-law behavior of the distribution tail; normal or log-normal distributions would have shown a strong drop-off appearance in the plot. When we use a least-squares regression of points in the plotted tail to estimate the heavy-tail α parameter[1], we find a value of 1.2. This value is consistent with our earlier observation of infinite variance; however, we cannot conclude that the distribution also has an infinite mean [16].

This observation of extreme variance at all time scales has significant importance for operating system engineering and tuning: Resource predictions are often made based on the observed mean and variance of resource requests, assuming that, over time, this will produce a stable system. Evidence from our traces shows that modeling the arrival rates of I/O request as a Poisson process or size distributions as a Normal distribution is incorrect. Using these simplified assumptions can lead to erroneous design and tuning decisions when systems are not prepared for extreme variance in input parameters, nor for the long-range dependence of system events.

An important reason for the departure from a normal distribution in file system analysis is that user behavior has a very reduced influence on most of the file system operations. Whether it is file size, file open times, inter-arrival rates of write operations, or directory poll operations, all of these are controlled through loops in the applications, through application defined overhead to user storage, or are based on input parameters outside of the user's direct control. More than 92% of the file accesses in our traces were from processes that take no direct user input, even though all the systems were used interactively. From those processes that do take user input, *explorer.exe*, the graphical user interface, is dominant, and although the user controls some of its operation, it is the structure and content of the file system that determines explorer's file system interactions, not the user requests. Unfortunately

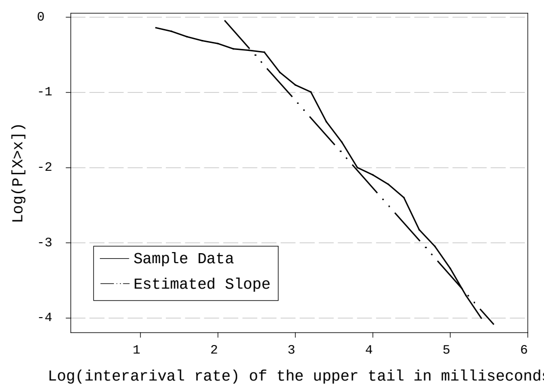

Figure 10: A log-log complementary distribution plot for the tail of the sample from figure 8, combined with a fitted line for the estimation of the α parameter

this kind of information cannot be extracted form the older traces so we cannot put this into a historical perspective.

This process controlled dominance of file system operations is similar to observations in data-communication, where, for example, the length of TCP sessions are process controlled, with only limited human factors involved. File system characteristics have an important impact on the network traffic; as for example the file size is a dominant factor in WWW session length. Given that the files and directories have heavy-tailed size distributions, this directly results into heavy-tailed distributions for those activities that depend on file system parameters [2,5].

Another important observation is that some characteristics of process activity, independent of the file system parameters, also play an important role in producing the heavy-tailed access characteristics. From the analysis of our traces we find that process lifetime, the number of dynamic loadable libraries accessed, the number of files open per process, and spacing of file accesses, all obey the characteristics of heavy-tail distributions. Some of these process characteristics cannot be seen as completely independent of the file system parameters; for example, the lifetime of the *winlogon* process is determined by the number and size of files in the user's profile.

Our observations of heavy-tail distributions in all areas of file system analysis lead to the following general conclusions:

1. We need to be very careful in describing file system characteristics using simple parameters such as average and variance, as they do not accurately describe the process of file system access. At minimum we need to describe results at different granularities and examine the data for extreme variances.
2. In our design of systems we need to be prepared for the heavy-tail characteristics of the access patterns. This is particularly important for the design and tuning of

[1] A random variable X follows a heavy-tailed distribution if $P[X > x] \sim x^{-\alpha}$, as $x \to \infty$, $0 < \alpha < 2$. A reliable estimator for α is the *Hill* estimator. We have computed this for our samples and it confirms the more llcd plot estimation results. A value of $\alpha < 2$ indicates infinite variance, if $\alpha < 1$ this also indicates an infinite mean.

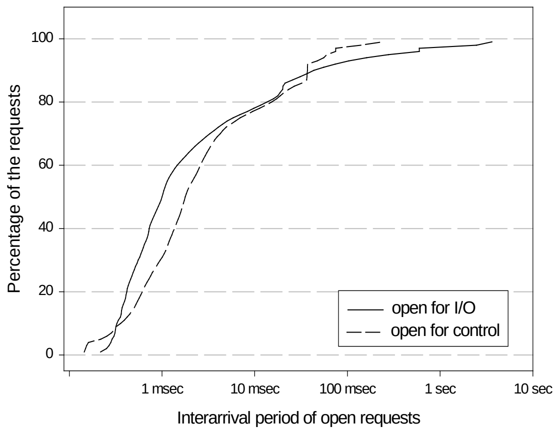

Figure 11. Cumulative distribution of the inter-arrival periods of file system open requests per usage type

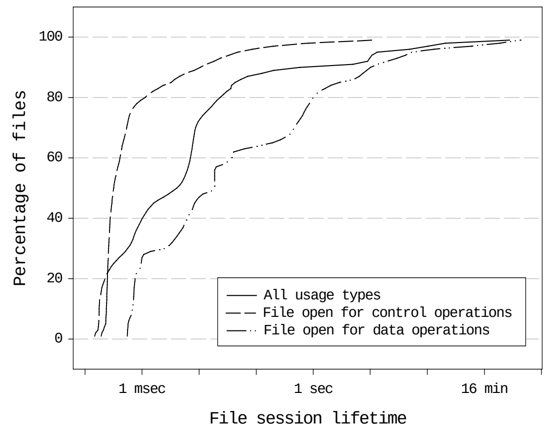

Figure 12. Cumulative distribution of the periods that files are open per usage type.

limited resource systems such as file caches, as there is important evidence that heavy-tail session length (such as open times and amount of bytes transferred) can easily lead to queue overflow and memory starvation [6].

3. When constructing synthetic workloads for use in file system design and benchmarking we need to ensure that the infinite variance characteristics are properly modeled in the file system test patterns. In [22], Seltzer et al. argue for application-specific file system benchmarking, which already allows more focused testing, but for each test application we need to ensure that the input parameters from the file system under test and the ON/OFF activity pattern of the application is modeled after the correct (heavy-tailed) distributions.
4. When using heuristics to model computer system operations it is of the highest importance to examine distributions for possible self-similar properties, which indicate high variance. Exploitation of these properties can lead to important improvements in the design of systems, as shown in [7].

8 Operational characteristics

There were 3 focus points when we analyzed the traces to understand the specifics of the Windows NT file system usage:

- Examine the traces from a system engineering perspective: the arrival rate of events, the holding time of resources, and the resource requests in general.
- Gain understanding in how applications use the functionality offered through the broad Windows NT file system interface and how the various options are exploited.
- Investigate the complexity of the Windows NT application and file system interactions.

In this section we explore these points by looking at the different file system operations, while in the next 2 sections we will investigate cache manager related technologies from these perspectives.

8.1 Open and close characteristics

Any sequence of operations on a file in Windows NT is encapsulated in an Open/Close sequence of events. Some operating systems have core primitives such as rename and delete which do not require the caller to open the file first, but in Windows NT these operations are generic file operations on files that have been opened first. For example, the deletion of a file or the loading of an executable can only be performed after the file itself has been opened.

Figure 11 displays inter-arrival times of open requests arriving at the file system: 40% of the requests arrive within 1 millisecond of a previous request, while 90% arrives with 30 milliseconds. When we investigate the arrivals by grouping them into intervals, we see that only up to 24% of the 1-second intervals of a user's session have open requests recorded for them. This again shows us the extreme burstiness of the system.

If we examine the reuse of files, we see that between 24% and 40% of the files that are opened read-only are opened multiple times during a user's session. Of the files accessed write-only, 4% are opened for another write-only session, while 36%-52% are re-opened for reading. 94% of the files that were open for reading and writing are opened multiple times, in the same mode.

An important measurement for resource tuning is the time that files are kept open (file session lifetime). In figure 12 we present the session lifetimes for a number of cases. The overall statistics show that 40% of the files are closed within one millisecond after they were opened and that 90% are open less then one second. Of the sessions with only control or directory operations 90% closed within 10 milliseconds.

When we investigate session times for the type of data access, we see that 70% of read-write access happens in periods of less then 1 second, while read-only and write-only accesses have this 1 second mark at 60% and 30%, respectively.

The session length can also be viewed from the process perspective. Some processes only have a single style of file access and the session time for each access is similar. The FrontPage HTML editor, for example, never keeps files open for longer then a few milliseconds. Others such as the development environments, databases control engines or the services control program keep 40%-50% of their files open for the complete duration of their lifetime. Programs such as *loadwc*, which manages a user's web subscription content, keep a large number of files open for the duration of the complete user session, which may be days or weeks. The first approach, opening a file only for the time necessary to complete IO, would produce a correlation between session time and file size. When testing our samples for such a correlation we could not find any evidence.

In general it is difficult to predict when a file is opened what the expected session time will be. All session distributions, however, had strong heavy-tails, from which we can conclude that once a file is open for a relatively long period (3-5 seconds, in most cases) the probability that the file will remain open for a very long time is significant.

Windows NT has a two stage close operation. At the close of the file handle by the process or kernel module, the IO manager sends a *cleanup* request down the chain of drivers, asking each driver to release all resources. In the case of a cached file, the cache manager and the VM manager still hold references to the FileObject, and the cleanup request is a signal for each manager to start releasing related resources. After the reference count reaches zero, the IO manager sends the *close* request to the drivers. In the case of read caching this happens immediately as we see the close request within 4-8 μsec after the cleanup request. In the case of write caching the references on the FileObject are released as soon as all the dirty pages have been written to disk, which may take 1-4 seconds.

8.2 Read and write characteristics

The burst behavior we saw at the level of file open requests has an even stronger presence at the level of the read and write requests. In 70% of the file opens, read/write actions were performed in batch form, and the file was closed again. Even in the case of files that are open longer than the read/write operations require, we see that the reads and writes to a file are clustered into sets of updates. In almost 80% of the reads, if the read was not at the end-of-file, a follow-up read will occur within 90 microseconds. Writes occur at an even faster pace: 80% have an inter-arrival space of less than 30 microseconds. The difference between read and write intervals is probably related to the fact that the application performs some processing after each read, while the writes are often pre-processed and written out in batch style.

When we examine the requests for the amount of data to be read or written, we find a distinct difference between the read and write requests. In 59% the read cases the request size is either 512 or 4096 bytes. Some of the common sizes are triggered by buffered file i/o of the *stdio* library. Of the remaining sizes, there is a strong preference for very small (2-8 bytes) and very large (48 Kbytes and higher) reads. The write sizes distribution is more diverse, especially in the lower bytes range (less then 1024 bytes), probably reflecting the writing of single data-structures.

8.3 Directory & control operations

The majority of file open requests are not made to read or write data. In 74%, the open session was established to perform a directory or a file control operation.

There are 33 major control operations on files available in Windows NT, with many operations having subdivisions using minor control codes. Most frequently used are the major control operations that test whether path, names, volumes and objects are valid. In general the application developer never requests these operations explicitly, but they are triggered by the Win32 runtime libraries. For example, a frequently arriving control operation is whether the "*volume is mounted*", which is issued in the name verification part of directory operations. This control operation is issued between up to 40 times a second on any reasonably active system.

Another frequently issued control operation is *SetEndOfFile*, which truncates the file to a given size. The cache manager always issues it before a file is closed that had data written to it. This is necessary as the delayed writes through the VM manager always have the size of one or more pages, and the last write to a page may write more data than there is in the file. The end-of-file operation then moves the end-of-file mark back to the correct position.

8.4 Errors

Not all operations are successful: of the open requests 12% fail and of the control operations 8% fail. In the open cases there are two major categories of errors: the file to be opened did not exist in 52% of the error cases and in 31% the creation of a file was requested, but it already did exist. When we examine the error cases more closely we see that a certain category of applications that uses the "open" request as a test for the existence of the file: the failure is immediately followed by a create action, which will be successful.

Reads hardly ever fail (0.2%); the error that does occur on the read are attempts to read past the end-of–file. We did not find any write errors.

9 The cache manager

An important aspect of the Windows NT file system design is the interaction with the cache manager. The Windows NT kernel is designed to be extensible with many third party software modules, including file systems, which forces the cache manager to provide generalized support for file caching. It also requires file system designers to be intimately familiar with the various interaction patterns between file system implementation, cache manager and virtual memory manager. A reasonably complete introduction can be found in [12].

In this section we will investigate two file system and cache manager interaction patterns: the read-ahead and lazy-write strategies for optimizing file caching. The cache manager never directly requests a file system to read or write data; it does this implicitly through the Virtual Memory system by creating memory-mapped sections of the files. Caching takes place at the logical file block level, not at the level of disk blocks.

A process can disable read caching for the file at file open time. This option is hardly ever used: read caching is disabled in only 0.2% of all files that had read/write actions performed on them.76% of those files were data files from opened by the "system" process. All of these files were used in a read-write pattern with a *write-through* option set to also disable write caching. Developers using this option need to be aware of the block size and alignment requirements of the underlying file system. All of the requests for these files will go through the traditional IRP path.

9.1 Read-ahead

When caching is initialized for a file, the Windows NT cache manager tries to predict application behavior and to initiate file system reads before the application requests the data, in order to improve cache hit rate. The standard granularity for read-ahead operation is 4096 bytes, but is under the control of the file system, which can change it on a per file basis. In many cases the FAT and NTFS file systems boost the read-ahead size to 65 Kbytes. Caching of a file is initiated when the first read or write request arrives at the file system driver.

Of all the sessions that performed reads 31% used a single IO operation to achieve their goal, and although this caused the caching to be initiated and data to be loaded in the cache, the cached data was never accessed after the first read.

Of the sequential accesses with multiple reads, which benefit from the read-ahead strategy, 40% used read sizes smaller than 4Kbytes and 92% smaller than 65Kbytes. This resulted in that only 8% of the read sequences required more than a single read-ahead action.

The cache manager tries to predict sequential access to a file so it can load data even more aggressively. If the application has specified at open time that the file data will be processed through sequential access only, the cache manager doubles the size of the read-ahead requests. Of file-opens with sequential read accesses only 5% specified this option. Of those files 99% were smaller than the read-ahead granularity and 80% smaller than a single page, so the option has no effect.

The cache manager also tries to predict sequential access by tracking the application actions: read-ahead is performed when the 3rd of a sequence of sequential requests arrives. In our traces this happened in 7% of the sequential cases that needed data beyond the initial read-ahead.

The cache manager uses a fuzzy notion of sequential access; when comparing requests, it masks the lowest 7 bits to allow some small gaps in the sequences. In our test in section 6.2, this would have increased the sequential marked trace runs by 1.5%.

9.2 Write-behind

Unless explicitly instructed by the application, the cache manager does not immediately write new data to disk. A number of lazy-write worker threads perform a scan of the cache every second, initiating the write to disk of a portion of the dirty pages, and requesting the close of a file after all references to the file object are released. The algorithm for the lazy-writing is complex and adaptive, and is outside of the scope of this paper. What is important to us is the bursts of write requests triggered by activity of the lazy-writer threads. In general, when the bursts occur, they are in groups of 2-8 requests, with sizes of one or more pages up to 65 Kbytes.

Applications have two methods for control over the write behavior of the cache. They can disable write caching at file open time, or they can request the cache manager to write its dirty pages to disk using a flush operation.

In 1.4% of file opens that had write operations posted to them, caching was disabled at open time. Of the files that were opened with write caching enabled, 4% actively controlled their caching by using the flush requests. The dominant strategy used by 87% of those applications was to flush after each write operation, which suggests they could have been more effective by disabling write caching at open time.

10 FastIO

For a long time the second access path over which requests arrived at the file system driver, dubbed the *FastIO* path, has been an undocumented part of the Windows NT kernel. The Device Driver Kit (DDK) documentation contains no references to this part of driver development, which is essential for the construction of file systems. The

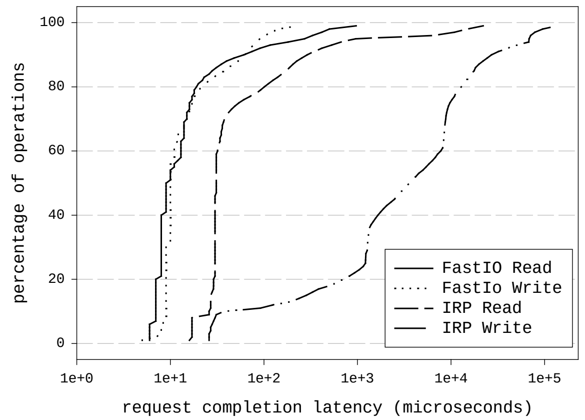

Figure 13. The cumulative distribution of the service period for each of the 4 major request types

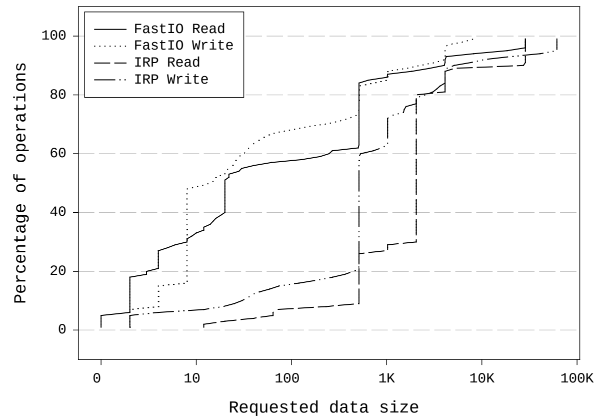

Figure 14. The cumulative distribution of the data request size for each of the 4 major request types

Installable File System Kit (IFS) shipped as Microsoft's official support for file system development, contains no documentation at all. Two recent books [12,20] provide some insight into the role of the FastIO path, but appear unaware of its key role in daily operations. In this section we will examine the importance of this access path, and provide some insight into its usage.

For some time the popular belief, triggered by the unwillingness of Microsoft to document FastIO, was that this path was a private "hack" of the Windows NT kernel developers to secretly bypass the general IO manager controlled IRP path. Although FastIO is a procedural interface, faster when compared with the message-passing interface of the IO manager, it is not an obscure hack. The "fast" in FastIO does not refer to the access path but to the fact that the routines provide a direct data path to the cache manager interface as used by the file systems. When file system drivers indicate that caching has been initialized for a file, the IO manager will try to transfer the data directly in and out of the cache by invoking methods from the FastIO interface. The IO manager does not invoke the cache manager directly but first allows file system filters and drivers to manipulate the request. If the request does not return a success value, the IO manager will in most cases retry the operation over the traditional IRP path. File system filter drivers that do not implement all of methods of the FastIO interface, not even as a passthrough operation, severely handicap the system by blocking the access of the IO manager to the FastIO interface of the underlying file system and thus to the cache manager.

Caching is not performed automatically for each file; a file system has to explicitly initialize caching for each individual file and in general a file system delays this until the first read or write request arrives. This results in a file access pattern where the traces will log a single read or write operation through the IRP interface, which sets up caching for that file, followed by a sequence of FastIO calls that interact with the file cache directly. The effect on latency of the different operations is shown in figure 13.

If we examine the size of the read requests in figure 14, we see that FastIO requests have a tendency towards smaller size. This is not related to the operation itself, but to the observation that processes that use multiple operations to read data, in general use more targeted sized buffers to achieve their goal. Processes that use only a few operations do this using larger buffers (page size, 4096 bytes, being the most popular).

Some processes takes this to the extreme; a non-Microsoft mailer uses a single 4Mbyte buffer to write to its files, while some of the Microsoft Java Tools read files in 2 and 4 byte sequences, often resulting in thousands of reads for a single class file.

The cache manager has functionality to avoid a copy of the data through a direct memory interface, providing improved read and write performance, and this functionality can be accessed through the IRP as well as the FastIO interface. We observed that only kernel-based services use this functionality.

11 Related work

File tracing has been an important tool for designing file systems and caches. There are 3 major tracing studies of general file systems: the BSD and Sprite studies [1,14], which were closely related and examined an academic environment. The 3rd study examined in detail the file usage under VMS at a number of commercial sites [15]. One of our goals was to examine the Windows NT traces from an operating system perspective; as such we compared our results with those found in the BSD and Sprite studies. The VMS study focused more on the differences between the various usage types encountered, and a comparison with our traces, although certainly interesting, was outside of the scope of this paper.

A number of other trace studies have been reported, however, they either focused on a specific target set, such as mobile users, or their results overlapped with the 3 major studies [3,9,11,23].

There is a significant body of work that focuses on specific subsets of file system usage, such as effective caching, or file system and storage system interaction.

There have been no previous reports on the tracing of file systems under Windows NT. A recent publication from researchers at Microsoft Research examines the content of Windows NT file systems, but does not report on trace-based usage [4].

With respect to our observations of heavy-tails in the distributions of our trace data samples; there is ample literature on this phenomenon, but little with respect to operation system research. A related area with recent studies is that of wide-area network traffic modeling and World Wide Web service models.

In [5], Gribble, et al. inspected a number of older traces, including the Sprite traces, for evidence of self-similarity and did indeed find such evidence for short, but not for long term behavior. They did conclude that the lack of detail in the older traces made the analysis very hard. The level of detail of the Windows NT traces is sufficient for this kind of analysis.

12 Summary

To examine file system usage we instrumented a collection of Windows NT 4.0 systems and traced, in detail, the interaction between processes and the file system. We compared the results of the traces with the results of the BSD and Sprite studies [1,14] performed in 1985 and 1991. A summary of our observations is presented in table 1.

We examined the samples for presence of heavy-tails in the distributions and for evidence of extreme variance. Our study confirmed the findings of others who examined smaller subsets of files: that files have a heavy-tail size distribution. But more importantly we encountered heavy-tails for almost all variables in our trace set: session inter-arrival time, session holding times, read/write frequencies, read/write buffer sizes, etc. This knowledge is of great importance to system engineering, tuning and benchmarking, and needs to be taken into account when designing systems that depend on distribution parameters.

When we examined the operational characteristics of the Windows NT file system we found further evidence of the extreme burstiness of the file systems events. We also saw that the complexity of the operation is mainly due to the large number of control operations issued and the interaction between the file systems, cache manager and virtual memory system.

The file system cache manager plays a crucial role in the overall file system operation. Because of the aggressive read-ahead and write-behind strategies, an amplification of the burstiness of file system requests occurs, this time triggered by the virtual memory system.

We examined the undocumented FastIO path and were able to shed light on its importance and its contribution to the overall Windows NT file system operation.

In this paper we reported on the first round of analysis of the collected trace data. There are many aspects of file system usage in Windows NT that have not been examined such as file sharing, file locking, details of the control operations, details of the various file cache access mechanisms, per process and per file type access characteristics, etc. We expect to report on this in the future.

Acknowledgements

Thanks goes to Fred Schneider for the motivation to do this study, to Thorsten von Eicken for the initial discussions of the results, and to my colleagues Ken Birman and Robbert van Renesse for never questioning what I was doing "on-the-side".

The paper improved significantly because of the very thoughtful comments of the reviewers and the help of Peter Chen, the official shepherd. Special praise goes to Sara Shaffzin, without whose editorial help this paper would never have made it through the first round. Tim Clark, Raoul Bhoedjang and Ken Birman provided useful help with the final version.

This investigation was made possible by grants from Microsoft Corporation and Intel Corporation, and was supported by the National Science Foundation under Grant No. EIA 97-03470 and by DARPA/ONR under contract N0014-96-1-10014.

References

[1] Baker, Mary G., John H. Hartmann, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, Measurement of a Distributed File System, in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198-212, Pacific Grove, CA, October 1991.

[2] Crovella, Mark, Murad Taqqu, Aze Bestevaros, "Heavy-Tailed Probability Distributions in the World Wide Web", in *A Practical Guide to Heavy-Tails: Statistical Techniques and Applications*, R. Adler, R. Feldman and M.S. Taqqu, Editors, 1998, Birkhauser Verlag, Cambridge, MA.

[3] Dahlin, Michael, Clifford Mather, Randolph Wang, Thomas Anderson, and David Patterson, A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 150 - 160, Nashville, TN, May 1994.

[4] Douceur, John, and William Bolosky, A Large Scale Study of File-System Contents, in *Proceedings of the SIGMETRICS'99 International Conference on Measurement and Modeling of Computer Systems*, pages 59-70, Atlanta, GA, May 1999.

[5] Gribble, Steven, Gurmeet SinghManku, Drew Roselli, Eric A.Brewer, Timothy J.Gibson, and Ethan L.Miller; Self-similarity in file systems , in *Proceedings of the SIGMETRICS'98 / PERFORMANCE'98 joint International Conference on Measurement and Modeling of Computer Systems*, pages 141 – 150, Madison, WI, June 1998.

[6] Heath, David, Sidney Resnick, and Gennady Samorodnitsky, *Patterns of Buffer Overflow in a Class of Queues with Long Memory in the Input Stream*, School of OR & IE technical report 1169, Cornell University, 1996.

[7] Harchol, Mor, and Allen Downey, Exploiting Process Lifetime Distributions for Dynamic Load Balancing, in *ACM Transactions on Computer Systems*, volume 15, number 3, pages 253-285, August 1997.

[8] Kistler, James J., and M. Satyanarayanan, Disconnected Operation in the Coda File System, in *ACM Transactions on Computer Systems*, volume 10, number 1, pages 3-25, February 1992.

[9] Kuenning, Geoffrey H., Gerald J. Popek, and Peter L. Reiher, An Analysis of Trace Data for Predictive File Caching in Computing, in *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 291-303, Boston, MA, June 1994.

[10] Majumdar, Shikharesh, and Richard B. Bunt, Measurement and Analysis of Locality Phases in File Referencing Behavior, in *Proceedings of the 1986 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 180-192, Raleigh, NC, May 1986.

[11] Mummert, Lily B., and M. Satyanarayanan, Long Term Distributed File Reference Tracing: Implementation and Experience, in *Software: Practice and Experience*, volume 26, number 6, pages 705-736, June 1996.

[12] Nagar, Rajeev, *Windows NT File System Internals*, O'Reilly & Associates, September 1997.

[13] Nelson, Michael N., Brent B. Welch, and John K. Ousterhout, Caching in the Sprite Network File System, *ACM Transactions on Computer Systems*, volume 6, number 1, pages 134-154, February 1988

[14] Ousterhout, John K., Herve Da Costa, David Harrison, John A. Kunze, Michael Kupfer and James G. Thompson, A Trace-Driven Analysis of the UNIX 4.2BSD File System, in *the Proceeding of the Tenth ACM Symposium on Operating Systems Principles*, pages 198-121, Orcas Island, WA, October 1991.

[15] Ramakrishnan, K. K., P. Biswas, and R. Karedla, Analysis of File I/O Traces in Commercial Computing Environments, in *Proceedings of the 1992 ACM SIGMETRICS and Performance '92 International Conference on Measurement and Modeling of Computer Systems*, pages 78-90, Pacific Grove, CA, June 1992.

[16] Resnick, Sidney I., *Heavy Tail Modeling and Teletraffic Data*, school of OR & IE technical report 1134, Cornell University, 1995.

[17] Rosenblum, Mendel, and John K.Ousterhout, The Design and Implementation of a Log-Structured File System, in *ACM Transactions on Computer Systems*. 10(1), pages 26–52, February 1992.

[18] Satyanarayanan, M., A Study of File Sizes and Functional Lifetimes, in *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 96-108, Pacific Grove, CA, December 1981.

[19] Smith, Alan Jay, Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering*, volume 7, number 4, pages 403-417, July 1981.

[20] Solomon, David, *Inside Windows NT, Second Edition*, Microsoft Press, 1998

[21] Willinger, Walt, and Vern Paxson, "Where Mathematics meets the Internet", in *Notices of the Amercian Mathematical Society*, volume 45, number 8, 1998

[22] Seltzer, Margo, David Krinsky, Keith Smith and Xiaolan Zhang, "The Case for Application-Specific Benchmarking", in *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems*, Rico, AZ, 1999

[23] Zhou, Songnian, Herve Da Costa and Alan Jay Smith, A File System Tracing Package for Berkeley UNIX, in *proceedings of the USENIX Summer 1985 Technical Conference*, pages 407-419, Portland Oregon, June 1985