Appears in Proceedings of the 2nd USENIX Windows NT Symposium, Seattle, WA, August 3-5, 1998.

A Performance Study of Sequential I/O on Windows NT™ 4

Erik Riedel
Carnegie Mellon University, riedel@cmu.edu
Catharine van Ingen, Jim Gray
Microsoft Research, {vanIngen,Gray}@microsoft.com

Abstract

Large-scale database, data mining, and multimedia applications require large, sequential transfers and have bandwidth as a key requirement. This paper investigates the performance of reading and writing large sequential files using the Windows NT™ 4.0 File System. The study explores the performance of Intel Pentium Pro™ based memory and IO subsystems, including the processor bus, the PCI bus, the SCSI bus, the disk controllers, and the disk media in a typical server or high-end desktop system. We provide details of the overhead costs at each level of the system and examine a variety of the available tuning knobs. We show that NTFS out-of-the-box performance is quite good, but overheads for small requests can be quite high. The best performance is achieved by using large requests, bypassing the file system cache, spreading the data across many disks and controllers, and using deep-asynchronous requests. This combination allows us to reach or exceed the half-power point of all the individual hardware components.

1 Introduction

High-speed sequential access is important for bulk data operations typically found in utility, multimedia, data mining, and scientific applications. High-speed sequential IO is also an important factor in the startup of interactive applications. Minimizing IO overhead and maximizing bandwidth frees power to actually process the data.

Figure 1 shows how data flows in a modern storage subsystem. Application requests are passed to the file system. If the file system cannot service the request from its main memory buffers, it passes requests to a host bus adapter (HBA) over a PCI peripheral bus. The HBA passes requests across the SCSI bus to the disk drive controller. The controller reads or writes the disk media and returns data via the reverse route.

The large, bold numbers in Figure 1 indicate the advertised throughputs listed on the boxes of the various hardware components. These are the figures quoted in hardware reviews and specifications. Several factors prevent you from achieving this PAP (peak advertised performance). The media-transfer speed and the processing power of the on-drive controller limit disk bandwidth. The wire's transfer rate, the disk transfer rate, and SCSI protocol overheads all limit throughput.

In the case diagrammed in Figure 1, the disk media is the bottleneck, limiting aggregate throughput to 7.2 MBps at each step of the pipeline. There is a significant gap between the advertised performance and this out-of-the-box performance. Moreover, the application consumes between 25% and 50% of the processor at this throughput. The processor would saturate long before it reached the advertised SCSI or PCI throughputs.

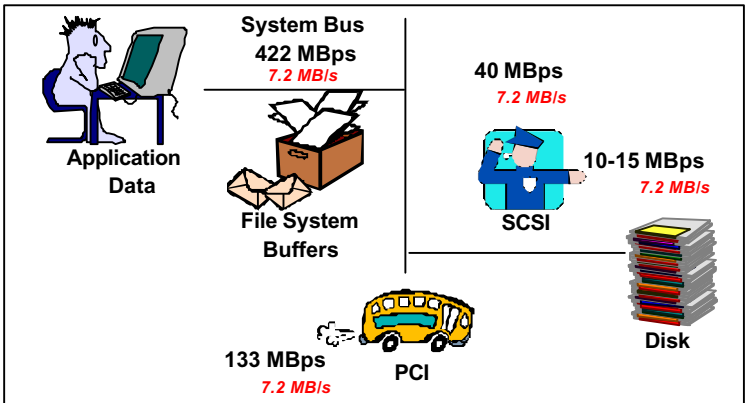

Figure 1 – The Storage Subsystem – An application makes requests of the file system, which transfers them across the PCI bus to a SCSI adapter that sends them across the SCSI bus to the disk. For each component, the upper numbers give the advertised speed and the lower number gives the actual speed in this application reading a single disk.

The goal of this study is to see if applications can do better cheaply - increase sequential IO throughput and decrease processor overhead while making as few application changes as possible. Our goal is to bring the real application performance (RAP) up to the *half-power point* - the point at which the system delivers at least half of the theoretical maximum performance. More succinctly, the goal is $RAP \geq PAP/2$. Such improvements often represent significant (2x to 10x) gains over the out-of-the-

box performance. We will see that the half-power point can be achieved without heroic effort, through a combination of techniques.

Our benchmark is a simple application that uses the NT file system to sequentially read and write a 100 MB file and times the result. `ReadFileEx()` and IO completion routines were used to keep *n* asynchronous requests in flight at all times. All measurements were repeated three times and, unless otherwise noted, all the data obtained were quite repeatable (within 3% margin of error). Multiple disk data was obtained by using NT *ftdisk* to build striped logical volumes and the basic system configuration used for all our measurements is described in Table 1.

The next section discusses our out-of-the-box measurements. Section 3 explores the basic capabilities of the hardware storage subsystem. Ways to improve performance by increasing parallelism are presented in Section 4. Section 5 provides more detailed discussion of performance limits and some additional software considerations. Finally, we summarize and suggest steps for additional study. An extended version of this paper, and all the benchmark software can be found at *www.research.microsoft.com/barc/Sequential_IO*.

2 Out-of-the-Box Performance

Our first measurements examine the out-of-the-box performance of our benchmark synchronously reading and writing using the NTFS defaults. The benchmark requests data sequentially from the file system. Since the data is not already in the file system cache, the file system fetches the data from disk into the cache and then copies it to the application's buffers. Similarly, when writing, the program's data is copied to the file cache and a separate thread asynchronously flushes the cache to disk in 64 KB units. In the out-of the-box experiments, the file being written was already allocated but not truncated. The program specified the `FILE_FLAG_SEQUENTIAL_SCAN` attribute when opening the file with `CreateFile()`. The total user and system processor time was measured via `GetProcessTimes()` and Figure 2 shows the results across a variety of application request sizes.

Buffered, sequential read throughput is nearly constant for request sizes up to 64 KB. The file system prefetches by issuing 64 KB requests to the disk. The disk controller also prefetches data from the media to its internal cache, which hides rotational delay and allows the disk to approach the media transfer limit. Figure 2 shows a sharp drop in read throughput for request sizes larger than 64 KB as the file system and disk prefetch mechanism fails (this problem is fixed in NT5). Figure 2 also indicates that buffered-sequential writes are substantially slower than reads. The file system performs write-back caching by default; it copies the contents of the application buffer into one or more file system buffers and the application considers the write complete when the copy is made. The file system then coalesces sequential requests into large 64 KB writes, leading to relatively constant throughput above 4 KB.

Disk controllers also implement write-through and write-back caching, controlled by the Write-Cache-Enable (WCE) setting directly at the device [SCSI93]. If WCE is disabled, the disk controller announces IO completion only after the media write is complete. If WCE is enabled, the disk announces write completion as soon as the data is stored in its cache and before the actual write onto the magnetic disk media. WCE allows the disk to hide the seek and media transfer, analogous to prefetching for reads. This improves write performance by giving pipeline parallelism – the write of the media overlaps the transfer of the next write on the SCSI even if the file system requests are synchronous. There is no standard default for WCE – a particular drive may be shipped with WCE enabled or disabled by default and a SCSI utility must be used to alter this setting. The effect of WCE can be dramatic as

Processor	Gateway 2000 G6-200, 200 MHz Pentium Pro, 1 32-bit PCI bus 64-bit wide 66 MHz memory interconnect, 64MB DRAM 4-way interleave							
Host bus adapter	1 or 2 Adaptec 2940UW Ultra-Wide SCSI adapters (40 MBps)							
Disk	Seagate Barracuda	Interface	Capacity	RPM	Seek	Transfer (MBps)		Cache
	Fast-Wide (ST15150W)	SCSI-2 FastWide	4.3 GB	7200	4.2ms	External 20 MBps	Internal 5.9 – 8.8	1 MB
	Ultra-Wide (ST34371W)	SCSI-2 UltraWide	4.3 GB	7200	4.2ms	40 MBps	10 - 15	512 KB
Software	Microsoft Windows NT Server 4.0 SP3, NT file system and NT's *ftdisk* for striping experiments							

Table 1 Basic Hardware and Software Configuration – This system is representative of a small server or high-end desktop system at the time these studies were performed in mid-1997.

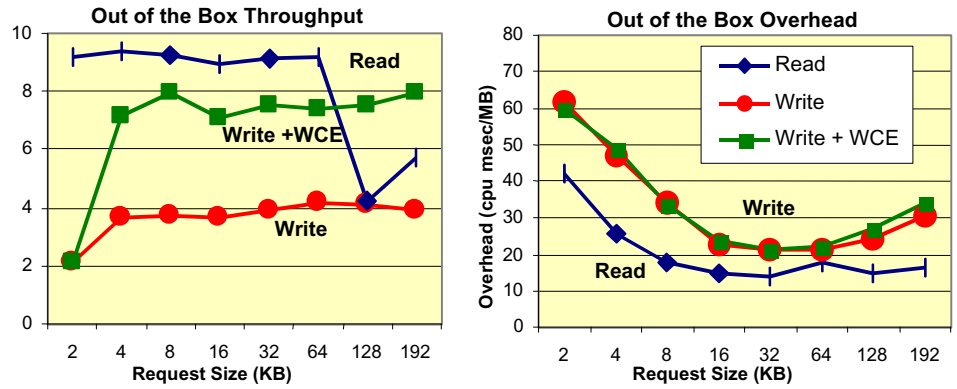

Figure 2 – Out-of-the-box Performance of a Single Ultra-Wide Drive – File system prefetching allows reads to reach full media bandwidth at small requests, although there is a sharp drop at very large request sizes. Using Write-Cache-Enable (WCE) nearly doubles write throughput. Processor cost per megabyte transferred shows that writes are more expensive than reads and overhead is minimal for requests in the 16 KB to 64 KB range.

shown in Figure 2 – WCE approximately doubles buffered sequential write throughput. When combined with file system write buffering, this allows small requests to attain throughput comparable to large request sizes and close to the performance of reads.[1]

Small requests involve many more system calls and protection domain crossings per megabyte of data moved. With 2 KB requests, the 200 MHz processor saturates when reading writing 16 MBps. With 64 KB requests, the same processor can generate about 50 MBps of buffered file IO – exceeding the Ultra-Wide SCSI PAP. As an upper bound, this processor and memory system can generate up to 480 MBps of unbuffered disk traffic.

Write requests of 2 KB present a particularly heavy load on the system. In this case, the filesystem must read the file prior to the write-back in units of 4 KB which more than doubles the load on the system. This pre-read can be avoided by (1) issuing write requests that are at least 4 KB, or (2) truncating the file at open by specifying `TRUNCATE_EXISTING` rather than `OPEN_EXISTING` as a parameter to `CreateFile()`. When we truncated the test file on open, throughput of 2 KB writes was about 3.7 MBps, just less than that of 4 KB and larger. `TRUNCATE_EXISTING` should only be used with small, buffered requests. With 4 KB and larger requests, extending the file after truncation incurs overheads which lower throughput up to 20%. This effect is discussed further in Section 5.3.

System behavior under large reads and writes is very different. During the read tests, processor load is fairly uniform. The file system prefetches data into the cache and then copies the data to the application's buffer. The file cache buffer can be reused as soon as the data is copied to the application. During the write tests, the processor load goes through three phases. In the first phase, the application writes at memory speed, saturating the processor as it fills all available file system buffers. During the second phase, the file system must free buffers by initiating SCSI transfers. New application writes are admitted as buffers become available. The processor is about 30% busy during this phase. At the end of this phase, the application closes the file and forces the file system to synchronously flush all remaining writes - one SCSI transfer at any time. During this third phase, the processor load is negligible.

Not all processing overhead is charged to the benchmark process in Figure 2. Despite some uncertainty in the measurements, the basic trend remains: moving data with many small requests costs significantly more than moving the same data with fewer larger requests. We will return to the cost question in more detail in the next section.

3 Improving Performance - Bypassing the File System Cache

Our next experiments bypass file system buffering to more closely examine the underlying hardware performance. This section provides data on both Fast-Wide (20 MBps) and Ultra-Wide (40 MBps) disks. The Ultra-Wide disk is the current generation of the Seagate Barracuda 4LP product line and the Fast-Wide disk is the previous generation. Figure 3 shows that the devices are capable of 30% of the PAP speeds. The input file is opened with `CreateFile(,…FILE_FLAG_NO_BUFFERING|FILE_FLAG_SEQUENTIAL_SCAN,…)` and the file system performs no prefetching, no

[1] Enabling WCE improves performance but risks corruption if the disk fails while uncommitted data is in its cache. The on-disk cache may also be lost by SCSI bus resets [SCSI93].

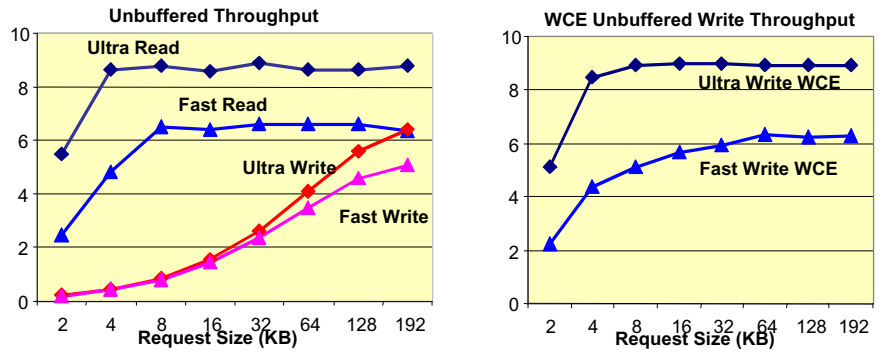

Figure 3 – Single Disk Throughput of Unbuffered IO for Fast-Wide and Ultra Drives – Requests of 8 KB and larger achieve the maximum read throughput. Write throughput is dramatically worse and increases only gradually because writes do not benefit from prefetching. The chart on the right shows that if drive write caching (WCE) is enabled, write throughput becomes comparable to read throughput. The newer Ultra drive has over a 100% advantage for small transfers, and a 50% advantage for large transfers due to its higher media transfer rate.

caching, no coalescing, and no extra copies. The data moves directly into the application from the SCSI adapter using direct memory access.

On large (64 KB) requests, bypassing the file system copy cuts the processor overhead by a factor of ten, from 2 instructions per byte to 0.2 instructions per byte. Unbuffered sequential reads reach the media limit for all requests larger than 8 KB. The older Fast-Wide disk requires read requests of 8 KB to reach its maximum efficiency of about 6.5 MBps. The newer Ultra-Wide drive plateaus at 8.5 MBps with 4 KB requests. Prefetching by the controller gives pipeline parallelism and allows drives to read at their media limits. Very large requests continue to perform at media rates, in contrast to the problems seen in Figure 2 with large buffered transfers.

Writes are significantly slower. The left chart of Figure 3 shows that throughput increases only gradually with request size. We observed no plateau in write throughput even for requests as large as 1 MB. The storage subsystem is completely synchronous – first it writes to the device cache, then to disk – so device overhead and latency dominate. Application requests above 64 KB are broken into multiple 64 KB requests in the IO subsystem, but these can be simultaneously outstanding at the device. The half-power write rate is achieved with a request size of 128 KB.

The right graph of Figure 3 shows that WCE compensates for the lack of file system coalescing. The WCE sequential write rates look similar to the read rates and the media limit is reached at about 8 KB for the newer disk and 64 KB for the older one. The media transfer time and rotational latency costs are hidden by the pipeline parallelism in the drive. WCE also allows the drive to perform fewer and larger media writes, reducing the total rotational latency.

Figure 4 shows the processor overhead corresponding to unbuffered sequential writes. In all cases, overheads

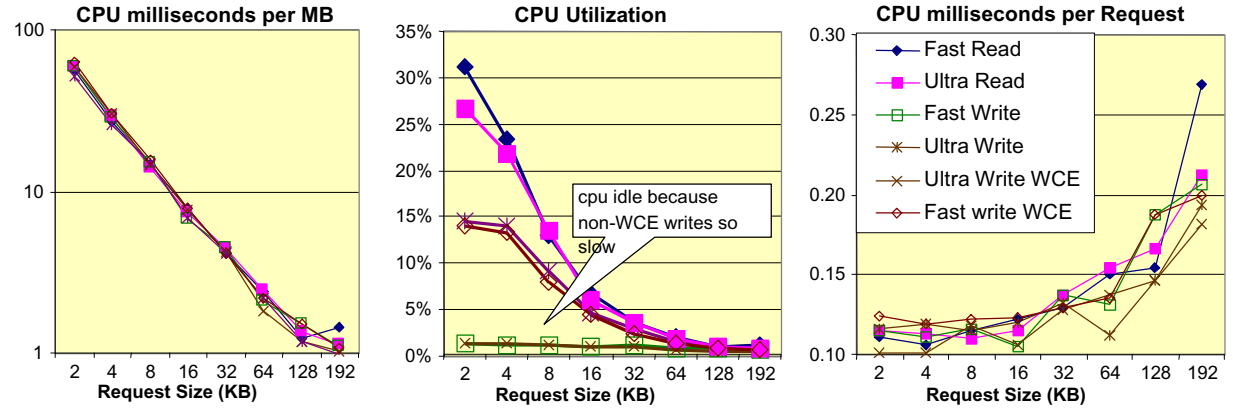

Figure 4 – Processing Cost of Unbuffered Sequential IO – The larger the request size, the more the cost of the request can be amortized. Requests of 64 KB reduce the load to less than 5%. Three drives running independent sequential streams of 2 KB requests would consume 96% of a 200 MHz Pentium Pro system.

decrease with request sizes. Requests less than 64 KB cost about 120µs and as requests become larger, the file system must do extra work to fragment them into 64 KB requests to the device. The first chart shows the processor time to transfer each megabyte of data. Issuing many small read requests places a heavy load on the processor while larger requests amortize the fixed overhead over many more bytes. The time is similar for reads and writes regardless of the generation of the disk and disk cache setting. The center chart of Figure 4 shows the processor utilization as a function of request size. At small requests, reads place a heavier load on the processor because the read throughput is so much higher than that of writes. The processor is doing approximately the same work per byte, but the bytes are moving faster so the imposed load is higher. Finally, the chart on the right of Figure 4 shows the processor time per request. Requests up to 16 KB consume approximately the same amount of time. Since a 16 KB request moves eight times as much data as a 2 KB request, we see a corresponding 8x change. Until the request size exceeds 64 KB, larger requests consume comparable processor time. Beyond 64 KB, the processor time increases because the file system does extra work, breaking the request into multiple 64 KB transfers and dynamically allocating control structures. Note that while the cost of a single request increases with request size, the cost per megabyte always decreases.

As a rule of thumb, requests cost about 120 µs, or about 10,000 instructions. Buffered requests have an additional cost of about 2 instructions per byte while unbuffered transfers have almost no marginal cost per byte. Recall that buffered IO saturates the processor at about 50 MBps for 64 KB requests. Unbuffered IO consumes about 2.1 ms per megabyte, so unbuffered IO will saturate this system's processor at about 480 MBps. On the system discussed here, the PCI peripheral bus would have become saturated long before this point and the memory bus would be near saturation.

4 Improving Performance via Parallelism

The previous sections examined the performance of synchronous requests to a single disk. Any parallelism in the system was due to caching by the file system or disk controller. This section examines two improvements: (1) using asynchronous IO to pipeline requests and (2) striping across multiple disks to allow media transfer parallelism.

Asynchronous IO increases throughput by providing the IO subsystem with more work to do at any instant. The disk and bus can overlap or pipeline the presented load and reduce idle time. As seen above, there is not much advantage to be gained by read parallelism on a single disk. The disk is already prefetching and additional outstanding requests create only a small additional overlap on the SCSI transfer. On the other hand, WCE parallelism dramatically improves single disk write performance.

In our asynchronous IO tests, the application issues multiple sequential IOs concurrently. When one re-

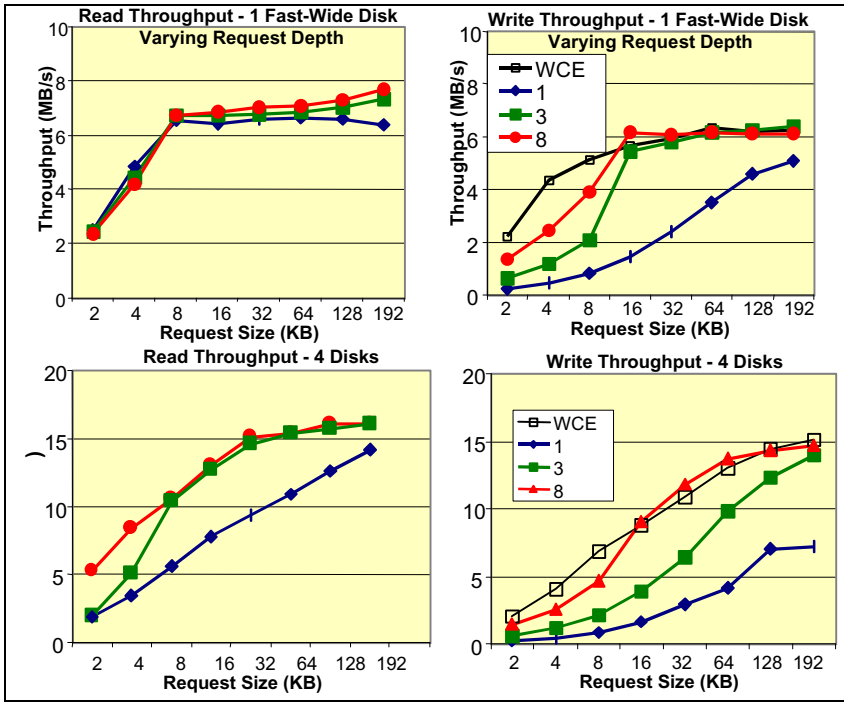

Figure 6 – Throughput of a Single Fast-Wide Drive and Four Disk Striping – The graphs show throughput for 1, 3, and 8 outstanding requests, as well as one outstanding request with WCE enabled. In the single disk case, asynchronous requests do not improve read performance because the disk controller is already prefetching. Asynchronous requests do improve write performance and, at larger request sizes, match the performance of enabling WCE. In the four drive case, asynchronous requests do improve read performance as the SCSI bus is better utilized, but writes are still substantially slower than reads.

quest completes, the application asynchronously issues another as part of the IO completion routine from the earlier request, attempting to keep *n* requests active at all times. The top of Figure 6 shows the read and write throughput of a single disk as the number of outstanding requests grows from 1 to 8. Read throughput is not much changed, while write throughput improves dramatically. Reads reach the half-power point with 4 KB requests. Writes need 3-deep 16 KB requests or 8-deep 8 KB requests to reach the half-power point, which represents a 4x improvement over synchronous 8 KB writes. For requests of 16 KB and more, 3-deep writes are comparable to the throughput of when WCE**.**

As more disks are added to the system, asynchronous IO gives significant benefits for reads and large transfers as well as smaller writes. The lower charts of Figure 6 show the results when the file is striped across four Fast-Wide SCSI disks on a single host bus adapter (and single SCSI bus). *ftdisk* is used to bind the drives into a stripe set and each successive disk gets the next 64 KB file chunk in round-robin fashion. At 4 KB and 8 KB requests, increasing request depth increases throughput as requests are spread across multiple disks. With a chunk size of 64 KB, 8-deep 8 KB requests will have IOs outstanding to more than one drive 7/8 of the time, approximately doubling the throughput. Smaller request depths distribute the load less effectively – with only two requests outstanding, IOs are outstanding to more than one drive only 1/4 of the time. Similarly, smaller request sizes are less effective since more requests are required for each stripe chunk. At 4 KB requests and 8 deep requests, at most two drives are used, and this only 3/8 of the time. Striping large requests improves the throughput of both reads and writes and the bottleneck moves from the disk media to the SCSI bus. Each disk can deliver about 6 MBps, so four disks should deliver up to 24 MBps. The experiments all saturated at about 16 MBps, so the RAP bandwidth of our Fast-Wide SCSI subsystem is 80% of the 20 MBps PAP. Ultra-Wide SCSI (not shown) also delivers 75% of PAP or about 30 MBps.

Both large request sizes and multiple disks are required to reach the SCSI bus half-power point. Fast-Wide SCSI can reach half-power points with two disks at read requests of 8 KB and write requests of 16 KB. Using 64 KB or larger requests, transfer rates up to 75% of the advertised bus bandwidth can be observed with three disks. Ultra-Wide SCSI reaches the half-power point with three disks and 16 KB read requests or 64 KB write requests. Only with very large reads can we reach 75% of the advertised bandwidth. The bus protocol overheads and actual data transfer rates do not scale with advertised bus speed. Further experiments show that three Ultra-Wide disks saturate a single Ultra-Wide SCSI bus. Two buses support a total of six disks and a maximum read throughput of 60 MBps. When a third adapter and three more disks are added, the PCI bus limit is reached and the configuration achieves a total of only 72 MBps – just over the half-power point of the PCI. Adding a fourth adapter shows no additional improvement, although the combined SCSI bandwidth of 120 MBps would seem to be well within the advertised 133 MBps of the PCI. While the practical limit is likely to depend on the exact hardware, the PCI half-power point appears to be a good goal.

5 Detailed Performance Measurements

The previous sections provided an overview of a typical storage system and discussed a number of parameters affecting sequential I/O throughput. This section investigates the hardware components in order to explain the observed behavior.

5.1 Disk Controller Caching and Prefetching

A simple model for the cost of a single disk read assumes no pipelining and separates the contributing factors:

$$\begin{aligned} Request_Service_Time = {} & Fixed_Service_Time \\ & + Disk_Seek_Time \\ & + (Transfer_Size / Media_Transfer_Rate) \\ & + (Request_Size / SCSI_TransferRate) \end{aligned}$$

The fixed overhead term includes time for the application to issue and complete the IO, the time to arbitrate and transfer control information on the SCSI bus, converting the target logical block to physical media location. The fixed time also includes the disk controller SCSI command handling, and any other processing common to any data transfer request. The next two terms are the time required to locate and move the data from the physical media into the drive cache. The final term the time required to transfer data from the disk cache over the SCSI bus.

The actual disk behavior is more complicated because controllers prefetch and cache data. The media-transfer and seek times can overlap the SCSI transfer time. When a SCSI request is satisfied from the disk cache, the seek time and some part of the fixed overhead is eliminated. Even without buffering, sequential transfers incur only short seek times. Large transfers can minimize rotational latency by reading the entire track – full-track transfers can start with the next sector to come under the read-write head.

At the extremes, some simplifications should occur. For small (2KB) requests, the fixed overhead dominates the transfer times (> 0.5 ms). For large (> 32 KB) requests, the media-transfer time (> 8 ms) dominates. The fixed overhead is amortized over a larger number of bytes and the SCSI transfer rate is faster (> 2x) than the media-transfer rate. We measured the fixed overhead component for three generations of Seagate drives: the Narrow 15150N, the Fast-Wide 15150W, and the Ultra-Wide 34371W. Table 3 shows the results. The cache hit data were obtained by reading the same disk blocks repeatedly. The prefetch hit column was obtained using the benchmark program to sequentially read a 100 MB file. To ensure that the prefetched data would be in the drive cache at all times, a delay was inserted between SCSI requests for those transfers marked with asterisks (*).

Table 3 – Variation across disk generation - The elapsed time in ms for a cache hit and prefetch hit of varying request sizes directly. Times are measured from an ASPI driver program issuing SCSI commands and bypassing the NT file system. For the large request sizes, the drive is given sufficient time between requests to ensure that the request is always satisfied from prefetch buffers and not limited by media transfer rates. Surprisingly, the cache hit times are always larger than the prefetch hit times.

	Narrow-ST15150N		Fast-Wide-ST15150W		Ultra-Wide-ST34371W	
Size	*Cache Hit*	*Prefetch Hit*	*Cache Hit*	*Prefetch Hit*	*Cache Hit*	*Prefetch Hit*
5K	0.96	0.56	0.93	0.59	8.14	0.30
1K	1.01	0.63	0.97	0.59	8.14	0.32
2K	1.11	0.75	1.02	0.58	8.14	0.34
4K	1.33	0.93	1.13	0.61	8.13	0.40
8K	1.75	1.38	1.36	0.86	8.13	0.51
16K	2.63	2.25	1.81	1.31*	8.13	0.74*
32K	4.35	3.93*	2.75	2.25*	8.13	1.22*
64K	16.50	7.30*	16.50	4.05*	8.15	2.15*

We expected that the cache hit case would be a simple way to measure fixed overhead. The data are already in the drive cache so no media operation is necessary. The results, however, tell a different story. The prefetch hit times are uniformly smaller than the cache hit times. The firmware appears to be optimized for prefetching – it takes longer to recognize the reread as a cache hit. In fact, the constant high cache hit times of the 34371W imply that this drive does not recognize the reread as a cache hit and rereads the same full track at each request. At 64 KB, the request spans tracks; the jump in the 15150 drive times may also be due to media rereads.

The prefetch hit data follow a simple fixed cost plus SCSI transfer model up through 8 KB request sizes. The SCSI transfer time was computed using the advertised bus rate. The 15150 drives (both Narrow and Fast-Wide) have fixed overhead of about 0.58 milliseconds; the 34371W drive (Ultra-Wide) has overhead of about 0.3 milliseconds.

At larger requests, no simple model applies. At 64KB, the computed SCSI transfer times do not account for the full prefetch hit time and the remainder is greater than the observed fixed overhead times. The media-transfer rate is not the limit because of the delay between requests. Without the delay, the measurements showed larger variation and the total time was not fully accountable to media transfer. The total time appears to be due to a combination of prefetch hit and new prefetch. A 64KB request may span up to three disk tracks and at least that many prefetch buffers. Whether or not the disk prefetches beyond the track necessary to satisfy the current request is unclear and likely to be implementation specific. Whether or not the disk can respond promptly to a new SCSI request when queuing a new prefetch is also unclear.

Intelligence and caching in the drive allows overlap and parallelism across requests so simple behavioral models no longer capture the behavior. Moreover, drive behavior changes significantly across implementations [Worthington95]. While the media-transfer limit remains a valid half-power point target for bulk file transfers, understanding smaller scale or smaller data set disk behavior seems difficult at best.

5.2 SCSI Bus Activity

We used a bus analyzer to measure SCSI bus activity. Table 4 summarizes the contribution of each protocol cycle type to the total bus utilization while reading the standard 100 MB file. Comparing the first two columns, small requests suffer from two disadvantages:

Small requests spend a lot of time on overhead. Half the bus utilization (30% of 60%) goes to setting up the transfer. There are eight individual 8KB requests for each 64KB request. This causes the increased arbitration, message, command and select phase times.

Table 4 – SCSI Activity by Phase - For 8KB requests, only 45% of the SCSI bus is data transfer (column 2). The balance goes to SELECT/RESELECT activity and parameter messaging. Larger requests make much more efficient use of the bus - for 64KB requests, utilization drops by half and data transfer makes up almost 90% of that time (column 3). When more disks are added, this efficiency drops somewhat in favor of more message traffic and SELECT activity. The three-disk system reaches over 99% bus utilization and consumes significantly more time in SELECT (column 5).

Phase	8KB Requests	64KB Requests		
	1 Disk	1 Disk	2 Disks	3 Disks
Arbitrate	1.1%	0.4%	0.6%	0.4%
Arbitrate Win	0.6%	0.2%	0.3%	0.2%
Reselect	0.2%	0.1%	0.1%	0.1%
Select	25.2%	0.2%	0.8%	4.4%
(Re)Select End	0.3%	0.1%	0.1%	0.1%
Message In	18.5%	7.4%	11.4%	9.1%
Message Out	5.5%	1.4%	2.8%	3.6%
Command	2.1%	0.5%	1.0%	1.1%
Data In	44.9%	89.3%	82.2%	80.4%
Data In End	0.7%	0.3%	0.4%	0.2%
Data Out	-	-	-	-
Data Out End	-	-	-	-
Status	0.7%	0.2%	0.3%	0.4%
Bus Utilization	**59.8%**	**30.1%**	**67.8%**	**99.3%**

Small requests spend little time transferring user data. At 64KB, 90% of the bus utilization is due to application data transfer. At 8KB, only 45% of the bus time is spent transferring application data.

The last two columns of Table 4 show the effects of SCSI bus contention. Adding a second disk doubles throughput but bus utilization increases 125%. The extra 25% is spent on increased handshaking (SELECT activity and parameter passing). The SCSI adapter is pending requests to the drives and must re-SELECT the drive when the request can be satisfied by the drive. More of the bus is consumed coordinating communication among the disks. Adding a third disk increases throughput and fully consumes the SCSI bus, as discussed in Section 3. The SELECT activity increases again, further reducing the time available for data transfer. The overall bus efficiency decreases as disks are added because more bus cycles are required coordinate among the drives.

5.3 Allocate

Unbuffered file writes have a serious performance pitfall. The NT file system forces unbuffered writes to be synchronous whenever a file is newly created and whenever the file is being extended either explicitly or by writing beyond the end of file. This synchronous write behavior also happens for files that are truncated (specifying the `TRUNCATE_EXISTING` attribute at `CreateFile()` or after open with `SetEndOfFile()`).

As illustrated in Figure 12, allocation severely impacts asynchronous IO performance. The file system allows only one request outstanding to the volume. If the access pattern is not sequential, the file system may actually zero any new blocks between requests in the extended region. Buffered sequential writes are not as severely affected, but still benefit from preallocation. Extending a file incurs at most about a 20% throughput penalty with small file system buffered writes. There is one notable exception. If you use tiny 2 KB requests, allowing the file system to allocate storage dynamically actually improves performance. The file system does not pre-read the data prior to attempting to coalesce writes.

To maximize asynchronous write performance, you should preallocate the file storage. If the space is not pre-allocated, the NT file system will first zero it be-

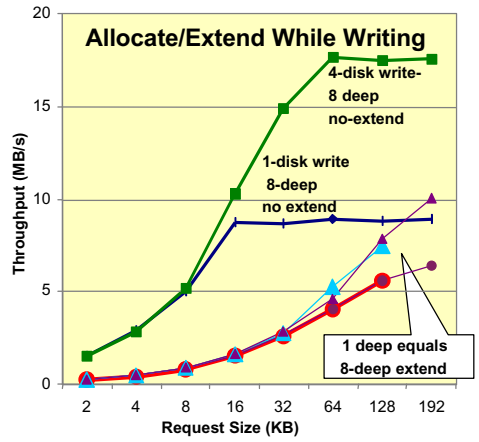

Figure 12 – File Allocate/Extend Behavior – When a file is being extended (new space allocated at the end), NT forces synchronous write behavior to prevent requests from arriving at the disk out-of-order. Security mandates that the value zero be returned to a reader of any byte which is allocated but has not yet been written. The file system must balance performance against the need to prevent programs from allocating files and then reading data from files deallocated by other users. The extra allocate writes dramatically slow write performance.

fore letting your program read it.

5.4 Alignment

The NT 4.0 file system (using the *ftdisk* mechanism) supports host-based software RAID 0, 1, and 5. A fixed stripe *chunk size* of 64 KB is used to build RAID0 stripe sets. Each successive disk gets the next 64KB chunk in round-robin fashion. The chunk size is not user-settable and is independent of the number or size of the stripe set components. The file system allocates file blocks in multiples of the file system *allocation unit* chosen when the volume is formatted. The allocation unit defaults to a value in the range of 512 bytes to 4 KB depending on the volume size. The stripe chunk and file system allocation units are totally independent; NT does not take the chunk size into account when allocating file blocks. Thus, files on a multiple-disk stripe set will almost always be misaligned with respect to stripe chunk size.

Figure 13 shows the effect of this misalignment. Alignment with the stripe chunk improves performance by 15-20% at 64 KB requests. A misaligned 64 KB application request causes two disk requests (one of 12 KB and another of 52 KB) that must both be serviced before the application request can complete. As shown earlier, splitting application requests into smaller units reduces drive efficiency. The drive array and host-bus adapter sees twice the number of requests and some of those requests are small. As the SCSI bus becomes loaded, the performance degradation becomes more noticeable. When requests are issued 8-deep, there are eight 64 KB requests active at any given time. In the misaligned case, there are 16 requests of mixed 12 KB and 52 KB sizes to be coordinated.

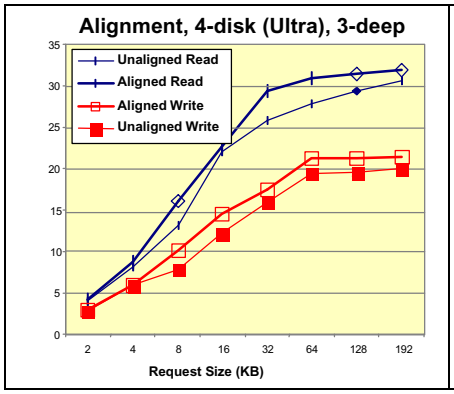

Figure 13 – Alignment Across Disks in a Stripe Set – The performance of a file aligned to the stripe chunk is compared to a file that is mis-aligned by 12KB. If requests split across stripe set step boundaries, read performance can be reduced by nearly 20% and writes by 15%. The effect is more pronounced with 8 requests outstanding because there is more activity on the SCSI bus and more contention.

Misalignment can be avoided by using the NT file system `format` command at the command prompt rather than the Disk Administrator application.[2] Disk Administrator limits the allocation size to 512, 1024, 2048, or 4096 bytes, while format command allows increments up to 64 KB. The cost of using a 64 KB allocation unit is the potential wasted disk space if the volume contains many small files; the file system always rounds the file size to the allocation unit.

6 Summary and Conclusions

The NT 4.0 file system out-of-the-box sequential IO performance is good: reads are close to the media limit and writes are near the half-power point. This performance comes at some cost; the file system is copying every byte, and coalescing disk requests into 64 KB units. Write throughput can be nearly doubled by enabling WCE, although this risks data corruption, and similar results can be achieved by using large requests and issuing asynchronous requests. NT file striping across multiple disks is an excellent way to increase throughput, but in order to take advantage of the available parallelism; striping must be combined with large and deep asynchronous requests.

An application can saturate a SCSI bus with three drives. By using multiple SCSI busses, it can saturate a PCI bus. By using multiple PCI buses, it could saturate the processor bus and memory subsystem. If the system configuration is balanced (disks do not saturate busses, busses do not saturate), the NT file system can reach the half-power point. In fact, applications can reach the sum of the device media limits by using a combination of (1) large request sizes, (2) deep asynchronous requests, (3) WCE, (4) striping, and (5) unbuffered IO.

Write performance is often significantly lower than read performance. The main pitfalls in writing files are: (1) if a file is not already allocated, the file system will force sequential writing in order to prevent applications from reading data left on disk by the previous file using that disk space (2) if a file is allocated but not truncated on open, then smaller than 4 KB buffered writes will first read a 4 KB unit and then overwrite (3) if the stripe chunk size is not aligned with the file system allocation size, large requests are broken into two smaller requests split across two drives, which doubles the number of requests to the drive array.

[2] A command of the form '`format e: /fs:ntfs /a:64k`' to create a file system with 64 KB allocation.

The measurements suggest a number of ways of doing efficient sequential file access:

- Larger requests are faster. Requests should be at least 8 KB, 64 KB if possible.
- Small requests consume significantly more processor time per byte than larger ones. 2 KB requests consume more than 30% of the processor while 64 KB requests both go faster and consume only 3% of the processor.
- If an application absolutely must make small requests, double buffering is not enough parallelism. There are noticeable gains through 8-deep requests.
- Write-Cache-Enable at drives provides significant benefits for small requests. Issuing three-deep asynchronous requests comes close to WCE performance for larger requests.
- Three disks can saturate a SCSI bus, whether Fast-Wide (15 MBps max) or Ultra-Wide (31 MBps max). Adding more disks than this to a single bus does not improve performance.
- File system buffering coalesces small requests into 64 KB disk requests for both reads and writes. This provides significant performance improvement for requests smaller than 64 KB.
- At 64 KB and larger requests, file system buffering degrades performance from the non-buffered case.
- When possible, files should be preallocated to their eventual maximum size.
- Extending a file while writing forces synchronization of the requests and significantly degrades performance.

This paper provided a basic tour of the parameters that affect sequential IO performance in NT and examined the hardware limitations at each stage in the IO pipeline. We have provided guidance on how applications can take advantage of the parallelism in the system and overlap requests for best performance. We have also shown that while out-of-the-box performance is reasonable for some workloads, there are a number of parameters that can make a factor of two to ten difference in overall throughput.

Many areas are not discussed here and merit further attention. Programs using asynchronous I/O have several options for managing asynchronous requests, including completion routines, events, completion ports, and multi-threading. Our benchmark uses completion routines in an otherwise single-threaded program and we have not explored the tradeoffs and overheads of using the other methods. This analysis focused on a single benchmark application issuing a single stream of sequential requests. A production system is likely to have several applications competing for storage resources. This complicates the model since the device array no longer sees a single sequential request stream.

7 Acknowledgements

Tom Barclay did the initial development of the `iostress` benchmark used in all these studies. Barry Nolte and Mike Parkes pointed out the importance of the allocate issue. Doug Treuting, Steve Mattos and others at Adaptec helped us understand SCSI details and the how the device drivers work. Bill Courtright, Stan Skelton, Richard Vanderbilt, Mark Regester of Symbios Logic generously loaned us an array, host adapters, and their expertise. Brad Waters, Wael Bahaa-El-Din, and Maurice Franklin shared their experience, results, tools, and hardware laboratory. They helped us understand NT performance issues and gave us feedback on our preliminary measurements. Will Dahli helped us understand NT configuration and measurement. Don Slutz and Joe Barrera gave us valuable comments, feedback and help in understanding NT internals. Finally, we thank the anonymous reviewers for their comments and our shepherd, Brad Chen, for his help on short notice.

8 References

[Custer92] Helen Custer, *Inside the WindowsNT™ File System*, ISBN 1-55615-481, Microsoft Press, 1992.

[Custer94] Helen Custer, *Inside the WindowsNT™ File System*, ISBN 1-55615-660, Microsoft Press, 1994.

[Nagar97] Rajeev Nagar, *Windows NT File System Internals: A Developer's Guide*, ISBN 1-565922-492, O'Reilly & Associates, 1997.

[Richter95] Jeffrey Richter, *Advanced Windows: The Developers Guide to the Win32™ API for WindowsNT™ 3.5 and Windows 95,* ISBN 1-55615-677-4, Microsoft Press, 1995.

[Schwaderer96] W. David Schwaderer, Andrew W. Wilson Jr. *Understanding I/O Subsystems, First Edition,* ISBN 0-9651911-0-9, Adaptec Press, Milpitas, CA, 1996.

[SCSI93] *ANSI X3T9.2 Rev 10L, 7-SEP-93*, American National Standards Institute, *www.ansi.org*.

[Seagate97] Seagate Technology "Barracuda Family Product Specification" *www.seagate.com*.

[Worthington95] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes, "On-Line Extraction of SCSI Disk Drive Parameters", *Proceedings of ACM Sigmetrics*, May 1995, pp. 146-156.