A Performance Study of Sequential IO on WindowsNT™ 4.0

Erik Riedel (CMU)
Catharine Van Ingen
Jim Gray

September 1997

Technical Report
MSR-TR-97-34

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

A Performance Study of Sequential I/O on Windows NT™ 4.0

Erik Riedel, Catharine van Ingen, Jim Gray

Microsoft Research
301 Howard Street
San Francisco, California, 94105
http://www.research.microsoft.com/barc
vanIngen@Microsoft.com, Gray@Microsoft.com, riedel+@cmu.edu

Abstract

This paper investigates the most efficient way to read and write 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. We provide details of the overhead costs at various levels of the system and examine a variety of the available tuning knobs. The report shows that NTFS out-of-the box read and write 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.

1. Introduction

This paper discusses how to do high-speed sequential file access using the Windows NT™ File System (NTFS). High-speed sequential file access is important for bulk data operations typically found in utility, multimedia, data mining, and scientific applications. High-speed sequential IO is also important in the startup of interactive applications. Minimizing IO overhead and maximizing bandwidth frees power to process the data.

Figure 1 shows how data flows in a typical storage sub-system doing sequential IO. 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 (Small Computer System Interconnect) bus to the disk drive controller. The controller reads or writes the disk and returns data via the reverse route.

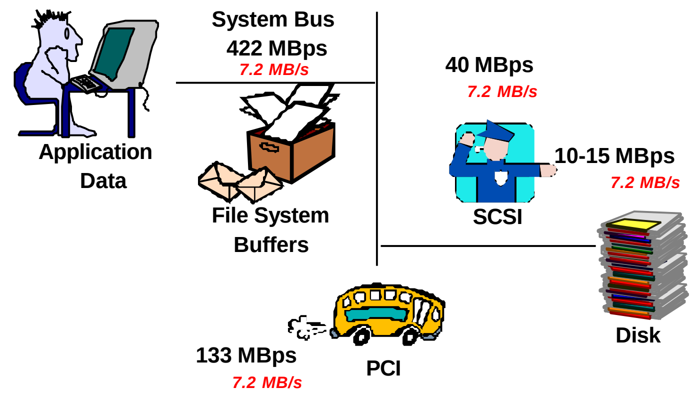


Figure 1 – The storage sub-system – An application makes requests of the file system, which transfers them across the PCI bus to a SCSI host-bus adapter that sends them across the SCSI bus to the disks(s). For each component, the upper numbers give the advertised speed, the lower number gives the actual speed in this application reading a single disk.

The large-bold numbers of Figure 1 indicate the advertised throughputs listed on the boxes of the various system 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 actual disk transfer rate, and SCSI protocol overheads ALL limit the throughput. The efficiency of a bus is the fraction of the bus cycles available for data transfer; in addition to data, bus cycles are consumed by contention, control transfers, device speed matching delays, and other device response delays. Similarly, PCI bus throughput is limited by its absolute speed, its protocol efficiency, and actual adapter performance. IO request processing overheads can also saturate the processor and limit the request rate.

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 IO pipeline. There is a significant gap between the advertised performance and this out-of-box performance. Moreover, the out-of-box application consumes between 25% and 50% of the processor. The processor would saturate before it reached the advertised SCSI throughput or PCI throughput.

The goal of this study is to do better cheaply - to increase sequential IO throughput and decrease processor overhead while making as few application changes as possible.

We define goodness as getting the real application performance (RAP) to the *half-power point* - the point at which the system delivers 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-box performance.

The half-power point can be achieved without heroic effort. The following techniques used independently or in combination can improve sequential IO performance.

Make larger requests: 8KB and 64KB IO requests give significantly higher throughput than smaller requests, and larger requests consume significantly less per-byte overhead at each point in the system.

Use file system buffers for small (<8KB) requests: The file system coalesces small sequential requests into large ones. It pipelines these requests to the IO subsystem in 64KB units. File system buffering consumes more processor overhead, but for small requests it can actually save processor time by reducing interrupts and reducing disk traffic.

Preallocate files to their eventual maximum size. Preallocation ensures that the file can be written with multiple requests outstanding (NT synchronously zeros newly allocated files). Preallocation also allows positioning the file on the media.

Write-Cache-Enable (WCE): Disks support write buffering in the controller. WCE allows the disk drive to coalesce and optimally schedule disk media writes, making bigger writes out of small write requests and giving the application pipeline-parallelism.

Stripe across multiple SCSI disks and buses: Adding disks increases bandwidth. Three disks can saturate the SCSI bus. To maximize sequential bandwidth, a SCSI host-bus adapter should be added for each three disks.

Unless otherwise noted, the system used for this study is the configuration described in Table 1:

Table 1 –All the measurements were done on the following hardware base (unless otherwise noted).

Processor	Gateway 2000 G6-200, 200 MHz Pentium Pro 64-bit wide 66Mhz memory interconnect 64MB DRAM 4-way interleave 1 32bit PCI bus							
Host bus adapter	1 or 2 Adaptec 2940UW Ultra-Wide SCSI adapters (40MBps)							
Disk	Seagate Barracuda 4	Interface	Capacity	RPM	Seek	Transfer		cache
						External	internal	
	Fast-Wide (ST15150W)	SCSI-2 Fast wide ASA II	4.3GB	7200	Avg 4.2ms range 1-17	20 MBps	5.9 MBps to 8.8 MBps	1 MB
	Ultra-Wide (ST34371W)	SCSI-2 Ultra wide ASA II	4.3GB	7200	Avg 4.2ms range 1-17	40MBps	10 MBps to 15 MBps	0.5MB
Software	Microsoft Windows NT 4.0 SP3 NT file system and NT's *ftdisk* for striping experiments.							

Table 1 suggests that the processor has a 422 MBps memory bus (66Mhz and 64-bit wide.) As shown later, this aggregate throughput is significantly more than that accessible to a single requestor (processor or PCI bus adapter). The study used SCSI-2 *Fast-Wide* (20MBps) and *Ultra-Wide* (40MBps) disks. As the paper is being written, *Ultra2* (80MBps) and *Fiber Channel* (100/200 MBps) disks are appearing.

The benchmark program is a simple application that uses the NT file system. It sequentially reads or writes a 100-MB file and times the result. `ReadFileEx()` and IO completion routines were used to keep ***n*** asynchronous requests in flight until the end of the file was reached; see the Appendix for more details on the program. Measurements were repeated three times. Unless otherwise noted, all the data obtained were quite repeatable (within 3%). All multiple disk data were obtained by using NT *ftdisk* to build striped logical volumes; *ftdisk* uses a stripe chunk, or step, size of 64KB. The program and the raw test results are available at http://www.research.microsoft.com/barc/Sequential_IO/.

The next section discusses our out-of-box measurements. Section 3 explores the basic capabilities of the hardware storage sub-system. Ways to improve performance by increasing parallelism are presented in Section 4. Section 5 provides more detailed discussion of performance limits at various points in the system and discusses some additional software considerations. Finally, we summarize and suggest steps for additional study.

2. Out-of-the-Box Performance

The first measurements examine the out-of-the-box performance of a program that synchronously reads or writes a sequential file using the NTFS defaults. In this experiment, the reading program requests data from the NT file system. The NT file system copies the data to the application request buffer from the main-memory file cache. If the requested data is not already in the buffer cache, the file system first fetches the data into cache from disk. When doing sequential scans, NT makes 64KB prefetch requests. Similarly, when writing, the program's data is copied to the NT file cache. A separate thread asynchronously flushes the cache to disk in 64KB transfer 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()`. Figure 2 shows the results.

Buffered-sequential read throughput is nearly constant for request sizes up to 64KB. The NT file system prefetches reads by issuing 64KB requests to the disk. The disk controller also prefetches data from the media to its internal controller cache. Depending on the firmware, the drive may prefetch only small requests by reading full media tracks or may perform more aggressive prefetch across tracks. Controller prefetching allows the disk to approach the media-transfer limit, and hides the disk's rotational delay. Figure 2 shows a sharp drop in read throughput for request sizes larger than 64KB; the NT file system and disk prefetch mechanisms are no longer working together.

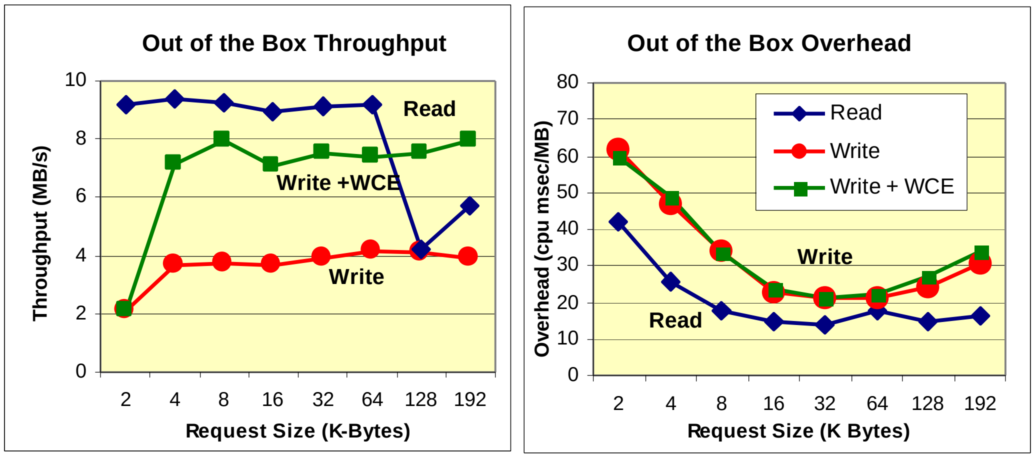


Figure 2 – Out-of-box Throughput of Ultra drives – File system pre-fetching causes reads to reach full media bandwidth at small request sizes, although there are difficulties at very large request sizes. Disk Write Cache Enable (WCE) approximately doubles sequential write throughput. Processor cost (milliseconds per megabyte) is graphed at the right. Writes are more expensive than reads, overhead is minimal for requests in the 16KB to 64KB range.

Figure 2 indicates that buffered-sequential writes are substantially slower than reads. The NT file system assumes write-back caching by default; the file system copies the contents of the application write buffer into one or more file system buffers. The application considers the buffered write completed when the copy is made. The file system coalesces small sequential writes into larger 64KB writes passed to the SCSI host bus adapter. The throughput is relatively constant above 4KB. The writeback occurs nearly synchronous – with one request outstanding at the disk drive. This ensures data integrity within the file. In the event of an error the file data are known to be good up to the failed request.

Write requests of 2KB present a particularly heavy load on the system. In this case, the filesystem reads the file prior to the write-back and those read requests are 4KB. This more than doubles the load on the system components. This pre-read can be avoided by (1) issuing write requests that are at least 4KB, or (2) truncating the file at open by specifying `TRUNCATE_EXISTING` rather than `OPEN_EXISTING` as the file creation parameter to `CreateFile()`. When we opened the test file with `TRUNCATE_EXISTING`, the write throughput of 2KB writes was about 3.7 MBps or just less than that of

4KB and above. `TRUNCATE_EXISTING` should be used with tiny, less than 4KB, buffered requests. With 4KB or larger requests, extending the file after truncation incurs overheads which lower throughput up to 20%.

The `FILE_FLAG_SEQUENTIAL_SCAN` flag had no visible affect on read performance, but improved write throughput by about 10%. Without the attribute, the write-back request size was no longer a constant 64KB, but rather varied between 16KB and 64KB. The smaller requests increased system load and decreased throughput.

The `FILE_FLAG_WRITE_THROUGH` flag has a catastrophic affect on write performance. The file system copies the application write buffer into the file system cache, but does not complete the request until the data have been written to media. Requests are not coalesced, the application request size is the SCSI bus request size. Moreover, the disk requests are completely synchronous – fewer writes complete per second. This causes almost a 10x reduction in throughput – with WCE and requests less than 64KB, we saw less than 1 MBps.

Disk controllers also implement write-through and write-back caching. This option is controlled by the Write-Cache-Enable (WCE) option [SCSI]. 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 IO completion as soon as the data are stored in its cache which may be before the actual write onto the magnetic disk media. WCE allows the disk to hide the rotational seek and media transfer. WCE 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 – the drive may come out of the box with WCE enabled or disabled.

The effect of WCE is dramatic. As shown in Figure 2 – WCE approximately doubles buffered-sequential write throughput. When combined with WCE, NT file-system write buffering allows small application request sizes to attain throughput comparable to large request sizes and comparable to read performance. In particular, it allows requests of 4KB or more to reach the half-power point.

Small requests involve many more NT calls and many more protection domain crossings per megabyte of data processed. With 2KB requests, the 200 MHz Intel Pentium processor saturates when reading writing 16 MBps. With 64KB requests, the same processor can generate about 50 MBps of buffered file IO – exceeding the Ultra-Wide SCSI PAP. As shown later, this processor can generate about 480 MBps of unbuffered disk traffic.

Figure 2 indicates that buffered reads consume less processing than buffered writes. Buffered writes were associated with more IO to the system disk, but we don't know how to interpret this observation.

To WCE or not to WCE?

Write caching improves performance but risks losing data if the caches are volatile. If the host or the disk controller fails while uncommitted data is in a non-volatile cache, that data will be lost. Also, controller caches may be lost by SCSI bus resets [SCSI]. Battery-backed RAM can be used to preserve a cache across power fails; some controllers commit pending data in their cache prior to responding to a SCSI reset. Typical UNIX semantics promise that a file writes will be written to non-volatile storage within 30 seconds. NT flushes its cache within 12 seconds or when the file is closed (unless it is a temporary file). The file system buffering risks are much higher than the disk cache risks since disk cache sector lifetimes are generally very short. Still, file systems and database systems assume that once a disk write completes, the data is safe. For that reason, it is dangerous to use WCE for database log files or for the NTFS directory space (both depend upon a write-ahead-logging scheme). There is always a risk of corrupting the volume if NTFS metadata is lost from the WCE cache. As shown in Figure 6, large asynchronous requests can match the throughput of WCE.

The system behavior under reads and writes is very different. During the read tests, the processor load is fairly uniform. The file system prefetches data to be read into the cache. It then copies the data from the file system cache to the application request buffer. The file cache buffer can be reused as soon as the data are copied to the application buffer. The elapsed time is about eleven seconds. During the write tests, the processor load goes through three phases. In the first phase, the application writes at memory copy 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 when buffers become available. The processor is about 30% busy during this phase. At the end of this phase the application closes the file. The close operation 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 process that caused it in Figure 2. Despite some uncertainty in the measurements, the 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.

In summary, the performance of a single-disk configuration is limited by the media transfer rate.

?? Reads are easy. For all request sizes, the out-of-box sequential-buffered-read performance achieves close to the media limit.

?? By default, small buffered-writes (less than 4KB) achieve 25% of the bandwidth. Buffered-sequential writes of 4KB or larger nearly achieve the half-power point.

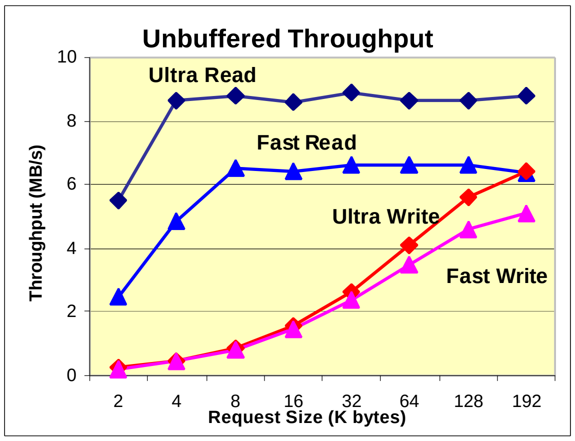
?? By enabling WCE, all but the smallest sequential buffered-write requests achieve 80% of the media-transfer limit.

?? For both reads and writes, larger request sizes have substantially less processor overhead per byte read or written. Minimal overhead occurs with requests between 16KB and 64KB.

3. Improving Performance - Bypassing the File System Cache for Large Requests

We next bypass file system buffering to examine the hardware performance. This section compares Fast-Wide (20MBps) and Ultra-Wide (40MBps) disks. Figure 3 shows that the devices are capable of 30% of the PAP speeds. The Ultra-Wide disk is the current generation of the Barracuda 4LP-product line (ST34371W). The Fast-Wide disk is the previous generation (ST15150W).

The 100MB file is opened with `CreateFile(,… FILE_FLAG_NO_BUFFERING | FILE_FLAG_SEQUENTIAL_SCAN,…)` to suppress file system buffering. The file system performs no prefetch, no caching, no coalescing, and no extra copies. The data moves directly into the application via the SCSI adapter using DMA (direct memory access). Application requests are presented directly to the SCSI adapter without first being copied to the file system buffer pool. On large (64KB) 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.



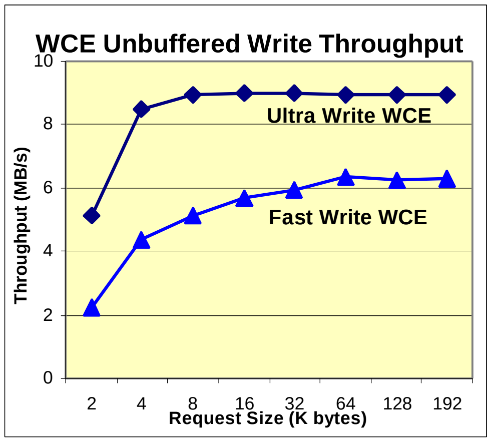


Figure 3 – Single Disk Throughput of Unbuffered IO for Fast-Wide and Ultra disks – The larger the request size, the higher the throughput. Requests above 8K achieve the maximum read throughput for a single disk. Write throughput is dramatically worse (left chart). Write throughput increases gradually because writes do not benefit from prefetching. The chart on the right shows that if disk-controller write caching is enabled (WCE), write throughput is comparable to read throughput, but the Fast-Wide drive requires larger requests to achieve full bandwidth. The newer Ultra drive has over a 100% advantage for small transfers, and a 50% advantage for large transfers due to the higher media transfer rate. Without WCE, sequential write behavior is horrible below 64KB requests.

Unbuffered-sequential reads reach the media limit for all requests larger than 8KB. The older Fast-Wide disk requires read requests of 8KB to reach its maximum efficiency of approximately 6.5 MBps. The newer Ultra-Wide drive plateaus at 8.5 MBps with 4KB requests. Prefetching by the controller gives pipeline parallelism allowing the drive to read at the media limit. Very large requests remain at the media limit (in contrast to the problems seen in Figure 2 with large buffered read transfers).

Without WCE, unbuffered-sequential writes are significantly slower. The left chart of Figure 3 shows that unbuffered-sequential write performance increases only gradually with request size. The differences between the two drives are primarily due to the difference in media density and drive electronics and not the SCSI bus speed. No write throughput plateau was observed even at 1MB request sizes. The storage subsystem is completely synchronous -- first it writes to device cache, then it writes to disk. Device overhead and latency dominate. Application requests above 64KB are still broken into multiple 64KB requests within the IO subsystem, but those requests can be simultaneously outstanding in the storage subsystem. Without WCE, the half-power write rate is achieved with a request size of 128KB.

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 8KB for the newer disk and 64KB for the older drive. The media-transfer time and rotational seek latency costs are hidden by the pipeline-parallelism of the WCE controller. WCE also allows the drive to perform fewer larger media writes, reducing the total rotational latency.

Figure 4 shows the processor overhead corresponding to the unbuffered sequential writes. Times are based on the total user and system time as reported by `GetProcessTimes()`. In all cases processor overheads decrease with request sizes. Requests less than 64KB appear to cost about 120?. As requests become larger, the file system has to do extra work to fragment them into 64KB requests to the device.

The first chart of Figure 4 shows the processor time to transfer a megabyte. Issuing many small read requests places a heavy load on the processor. Larger requests amortize the fixed overhead over many more bytes. The time is very similar for both reads and writes regardless of the generation of the disk and disk caching. The drive response time makes little difference to the host. With 2KB requests, this system can only generate a request rate of about 16 MBps.

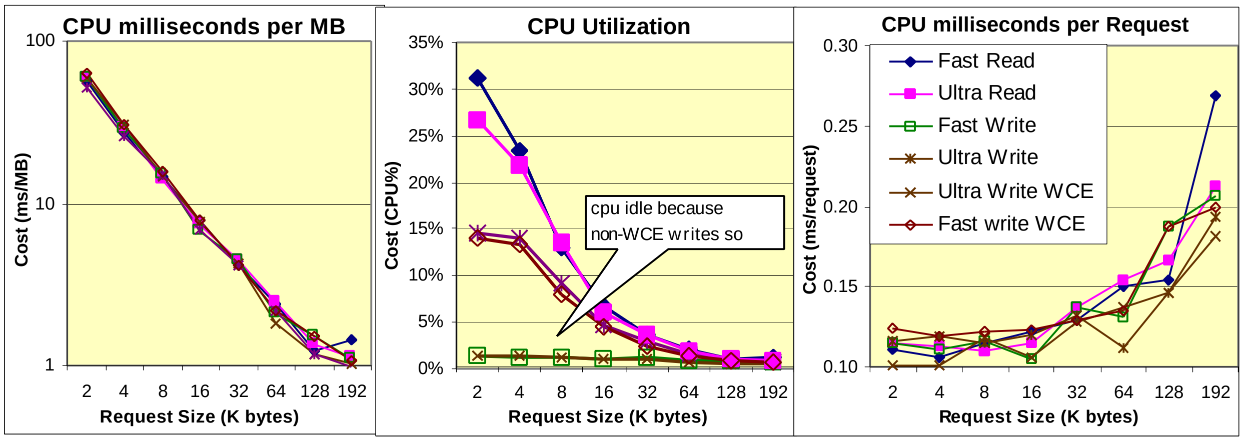


Figure 4 – Processing Cost of Unbuffered Sequential IO – The larger the request size, the more the cost of the request can be amortized. Requests of 64KB are necessary to reduce the load to less than 5%. Three drives running independent sequential streams of 2KB requests would consume 96% of a 200 MHz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

The middle chart of Figure 4 shows the host processor utilization as a function of request size. At small requests, reads place a heavier load on the processor because the read throughput is higher than that of writes. The processor is doing the same work per byte moved, but the bytes are moving faster so the imposed load is higher. Without WCE, write requests appear to place a much smaller load on the processor because sequential writes run much more slowly.

Finally, the chart on the right of Figure 4 shows the processor time per request. Requests up to 16KB consume approximately the same amount of processor time. Since the 16KB request moves eight times as much data as the 2KB request, we see the corresponding 8x change in the center chart. Until the request sizes exceed 64KB, larger requests consume comparable processor time. Beyond 64KB, the processor time increases because the file subsystem does extra work, breaking the request into multiple 64KB transfers, and dynamically allocating control structures. Note, however, that while the cost of a single request increases with request size, the processor cost per megabyte always decreases.

As a rule of thumb, requests cost about 120? seconds, 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 64KB requests. Unbuffered IO consumes about 2.1 milliseconds per megabyte, so unbuffered IO will saturate this processor at about 480 MBps. On the systems discussed here, the two PCI buses would have become saturated long before that point and the memory bus would be near saturation leaving no ability for the processor to process data.

Figure 5 summarizes the performance of an Ultra-Wide SCSI for Figures 2, 3, and 4 – a very busy graph. Buffered requests give significantly better performance below 4KB. Processing many small requests not only incurs more processor overhead but also reduces the efficiency of the disk controller. Above 16KB, the overhead of unbuffered reads is significantly (2-10x) less than buffered reads. Above 64KB, unbuffered reads are definitely better in both throughput and overhead.

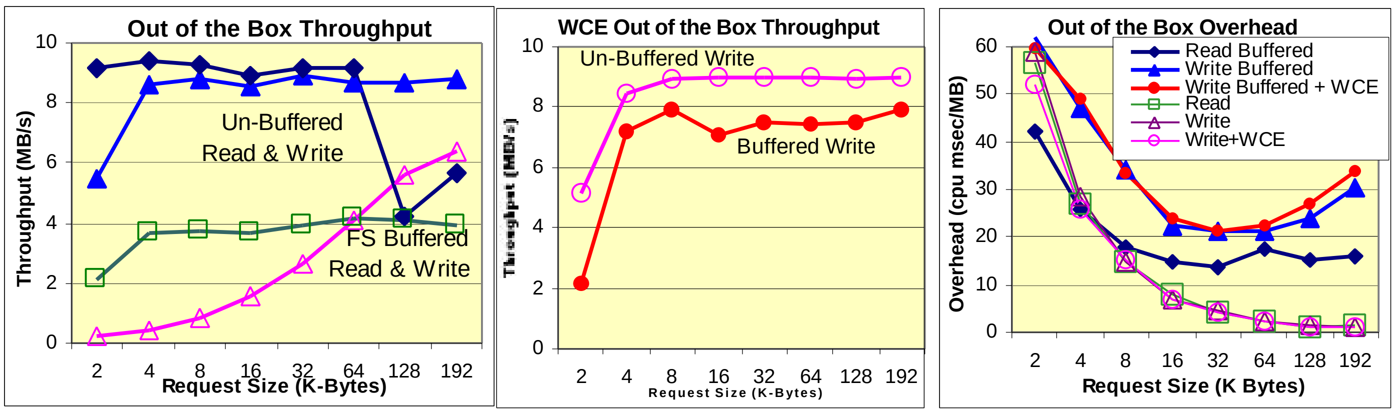


Figure 5 – Ultra drive performance – These charts combine Figures 2, 3 and 4 for an Ultra-SCSI drive. Small buffered reads show a double benefit: greatly increased throughput and slightly less processor cost. Large buffered reads have significant processor cost and no performance benefit. Without WCE, buffered writes have superior performance until 64KB, but incur large processor costs due to data copying for filesystem caching. Enabling WCE always improves throughput. The Overhead graph shows that buffering has high processor overhead for writes and for large reads.

Either WCE or file system buffering is necessary to achieve good sequential-write throughput. The best write performance occurs with large request sizes, WCE, and no file system buffering. Only with WCE or very large (128KB) requests can the disk reach the half-power point while writing.

To summarize:

?? Disk read prefetch makes it easy to achieve good sequential read performance.
?? Without WCE writes are much slower than reads.
?? Disk write-cache-enable (WCE) has performance benefits ranging from 10x (small-unbuffered writes) to 2x (large-buffered writes).
?? File system buffering gives a clear benefit for sequential write requests smaller than 64KB and for sequential reads smaller than 8KB.
?? Bypassing file system buffering for requests larger than 32KB dramatically reduces processor overheads.
?? A read or write request consumes about 10,000 instructions.
?? File system buffering consumes about 2 instructions per byte while unbuffered requests have almost no marginal cost per byte.

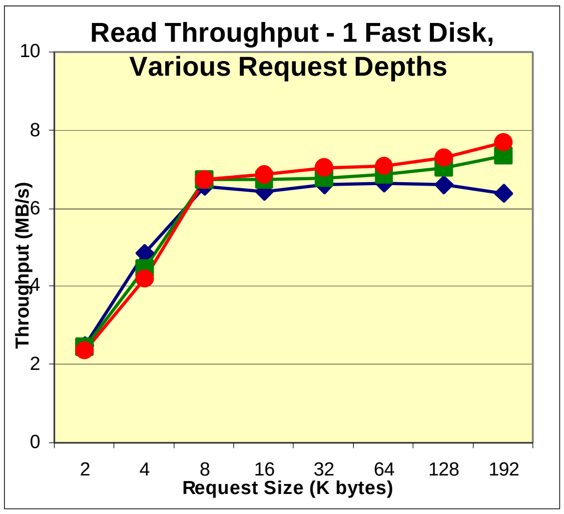
4. Improving Performance via Parallelism

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 throughput improvements: (1) using asynchronous IO to pipeline requests and (2) striping files across multiple disks and busses 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 busses can overlap or pipeline the presented load. This reduces idle time. As seen before, there is not much advantage to be gained by read parallelism on a single disk. The disk is already pipelining requests by prefetching; additional outstanding requests create a little additional overlap on the SCSI transfer. On the other hand, WCE's pipeline parallelism dramatically improves single-disk write performance. As you might expect, by issuing many unbuffered IO requests in parallel, the application can approximate the single-disk performance of WCE.

Figure 21 has the program details, but the idea is that the application issues multiple sequential IOs. When one IO completes, the application asynchronously issues another IO as part of the IO completion routine. The application attempts to keep ***n*** requests active at all times.

Figure 6 shows the read throughput of a single disk as the number of outstanding requests (request depth) grows from 1 to 8; the second chart shows write throughput. The results are as expected, read throughput is not much improved, write throughput improved dramatically. Read throughput always reaches the half-power point for 4KB requests. Writes need 3-deep 16KB requests or 8-deep 8KB requests to reach the half-power point. This is a 4x-performance improvement for 8KB writes. For request sizes of 16KB or more, 3-deep write throughput compares to the throughput of WCE**.**



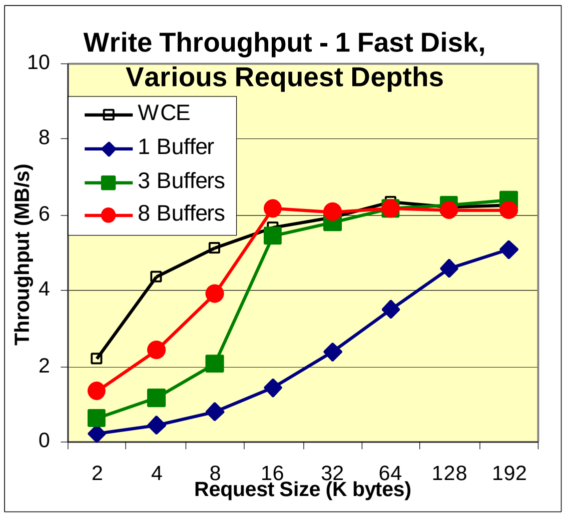
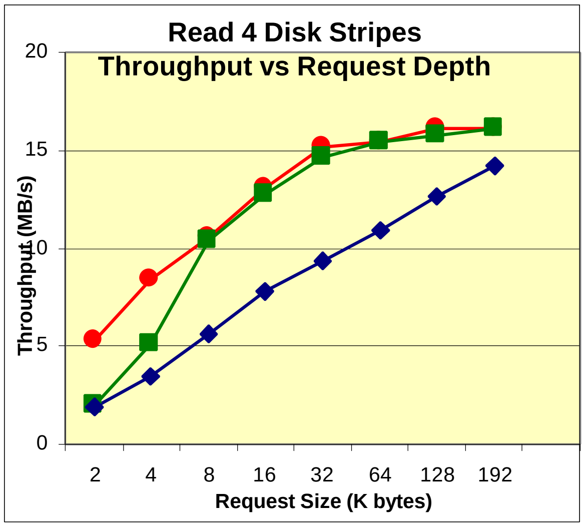


Figure 6 –Throughput of a Fast-Wide SCSI disk using request pipelining and no file system buffering. The graphs show sequential disk throughput for 1, 3, and 8 outstanding requests. Asynchronous requests do not improve read performance because the disk controller is already prefetching. Asynchronous requests do improve write performance. Indeed, at large request sizes, asynchronous requests match the performance of WCE.

Deep asynchronous IO performs as well as WCE - in both cases the disk can overlap and pipeline work. At sizes less than 16KB, WCE is more effective than asynchronous IO because the disk more effectively coalesces writes prior to media access and so performs fewer, larger writes. Without WCE, the disk must perform each write operation prior to retiring the command. This causes larger overhead in all parts of the IO pipeline and incurs more rotational delays. At sizes of 16KB and above, the media becomes the limit and WCE is no longer important. For these larger request sizes the WCE curve is comparable to the others.

Asynchronous IO gives significant benefit for reads and large transfers as well as smaller writes when more disks are added to the system. Figure 7 shows the results when the file is striped across four Fast-Wide SCSI disks on one host-bus adapter (one SCSI bus). NT *ftdisk* was used to bind the drives into a stripe set. . Each successive disk gets the next 64KB file chunk in round-robin fashion.



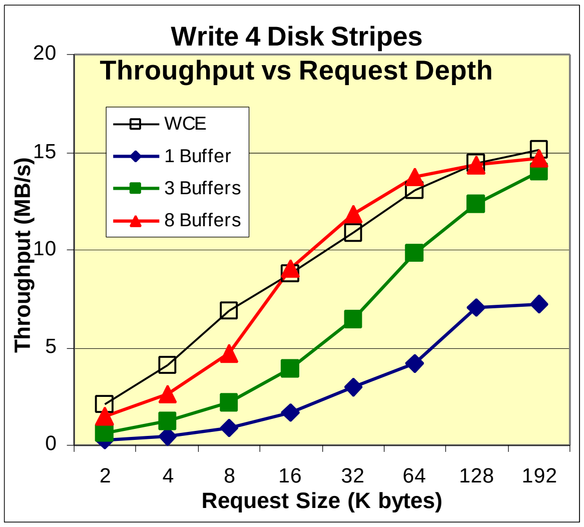


Figure 7 –Throughput of reads and writes striped across four Fast-Wide SCSI disks on one controller using request pipelining and no file system caching. The graphs show sequential disk throughput for 1, 3, and 8 outstanding requests. Asynchronous requests do improve read performance SCSI bus is better utilized. Writes are still substantially slower than reads. At large request sizes multiple outstanding requests has throughput comparable to WCE.

With 4KB and 8KB requests, increasing request depth increases throughput. This is because requests are being spread across multiple disks. Since the stripe chunk size is 64KB, 8-deep 8KB requests will have requests outstanding to more than one drive about 7/8 of the time. That almost evenly distributes requests across pairs of drives, approximately doubling the throughput. Smaller request depths distribute the load less effectively; with only two requests outstanding, requests are outstanding to more than one drive only about 1/4 of the time. Similarly, smaller request size distributes the load less effectively since more requests are required for each stripe chunk. With 4KB requests and 8 deep requests, at most two drives are used, and that only happens about 3/8 of the time.

Striping large requests improves the throughput of both reads and writes. At larger request sizes, the bottleneck moves from the disk media to the SCSI bus. Each disk can deliver about 6MBps, so four disks might deliver 24MBps. The experiments all saturated at about 16MBps. The RAP bandwidth of our Fast-Wide subsystem is 80% of the 20MBps PAP. Ultra-Wide SCSI (not shown) also delivers 75% of PAP or about 30 MBps.

Figure 8 examines the SCSI bus throughput as disks are added to a single bus and adapter. A request depth of three was used to access stripe sets of two, three, and four disks on a single bus. The rightmost chart shows the Ultra-SCSI write behavior. All write data were taken with WCE enabled. The processor overhead is not shown because the extra cost of stripping was negligible.

Adding a second disk nearly doubles throughput. Adding a third disk gives slightly less than linear scaling, while adding a fourth disk makes little improvement. In fact, with Ultra-Wide SCSI, adding the third disk makes no difference when writing; reads (not shown) show linear scaling up to three disks. Increasing the request depth (also not shown) causes the scaling to occur at slightly smaller request size, but the net effect is the same.

The observed limiting throughput for the Fast-Wide SCSI is about 16MBps. Ultra-Wide shows different limits: 30MBps for reads but only 20MBps for writes (see Figure 13). Three disks at 6MBps Fast-Wide or 10MBps Ultra-Wide reach the limit in both cases. (Again, note that the single disk performance varies somewhat between the two due to changes to the disk internals across drive generations as seen in Figure 3.)

Both large request size and multiple disks are required to reach the SCSI bus half-power point. The Fast-Wide SCSI can reach half-power points with two disks – the media speed is only half the bus speed. Read requests of 8KB and write requests of 16KB are needed. Using 64KB or larger requests, transfer rates of 75% of the advertised bus bandwidth can be observed with three disks. The Ultra-Wide SCSI reaches the half-power point with three disks and 16KB read requests or 64KB 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.

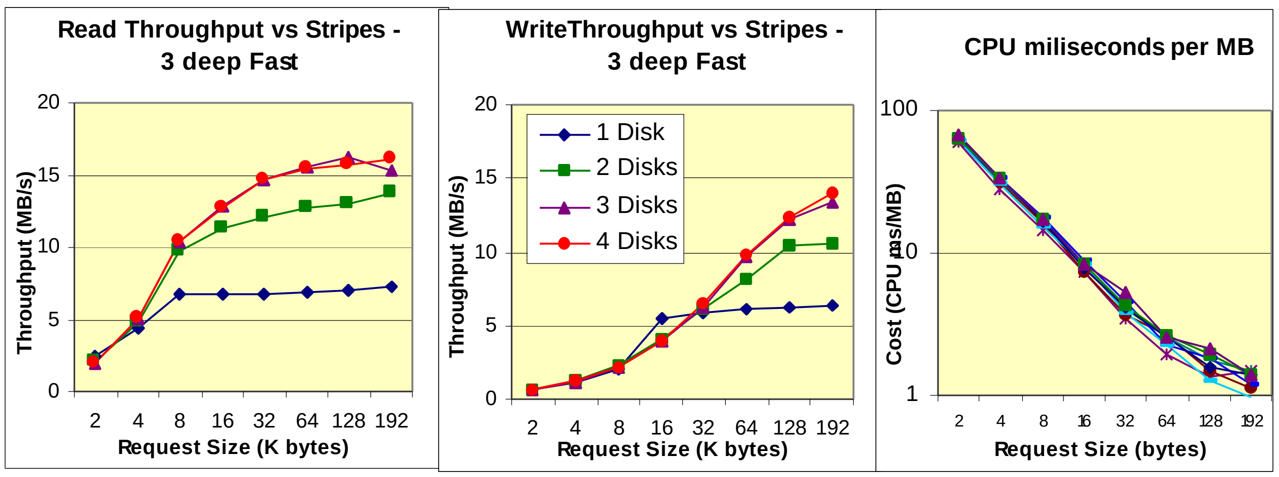


Figure 8 –Throughput vs. disk striping for a three-deep Fast-Wide SCSI array. As disks are added to the strip set, throughput increases until the SCSI bus saturates. Fast Wide is advertised at 20MBps but delivers about 16MBps. Ultra-Wide is advertised at 40MBps but saturates at about 30MBps for reads (not shown) and 20MBps for writes (see Figures 10, 13). WCE is enabled in this experiment, but even so, writes have consistently lower throughput than reads.

To benefit from extra disks, additional SCSI bus and host-bus adapters must be added. Figure 9 compares the throughput of four disks distributed across two buses (2 disks on each of 2 adapters) to that of four disks on a single bus. The two-bus two-adapter configuration continues to gain throughput with the fourth disk. For larger request sizes, the distributed 2+2 configuration gives nearly 24 MBps or double the 2 disk throughput. Adding the second adapter allows simultaneous transfers on both SCSI buses and allows more efficient use of the disks.

The additional parallelism across the SCSI buses may be limited by the stripe chunk size. To benefit from multiple buses, requests must be outstanding to drives on them.[1] The throughput with 2KB and 4KB requests is almost unchanged across the two configurations. Most of the time, only one disk has pending requests. With 8KB requests, requests are outstanding to two drives between 1/3 and 1/4 of the time. Whenever those drives are on different SCSI buses the resulting data transfers can occur simultaneously. That occurs half of the time with the 2+2 configuration. In other words, SCSI bus transfers could occur on both buses in parallel about 15% the time. To get the full benefit from the parallel buses, the application should have multiples of 64KB of IO outstanding on each bus.

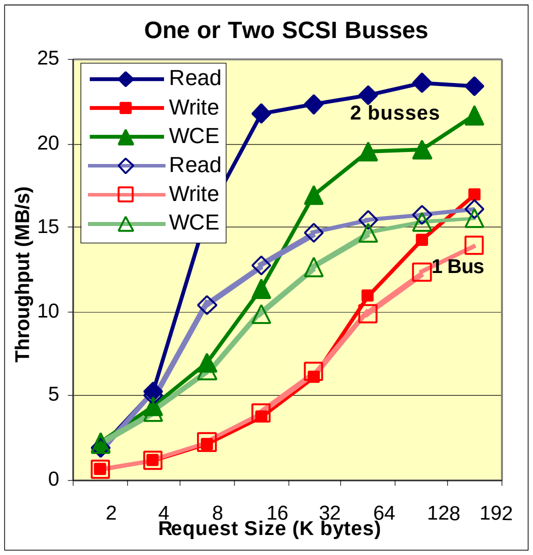


Figure 9 –Four fast disks on one or two SCSI busses. Parallel busses add bandwidth and so avoid bus saturation. Each disk can run at full speed when they are spread across two busses. Reading and writing is done with three-deep asynchronous requests. WCE is enabled on the drives. Each disk can deliver about 6 MBps, so four disks should be able to deliver 24MBps. One Fast-Wide SCSI bus saturates at 16MBps. When the disks are split between two busses, throughput approaches the expected 24MBps. If one more disk is added to each controller, peak throughput increases to 32MBps. Ultra-SCSI busses show similar behavior, except that each bus can carry twice the bandwidth (30MBps) and the newer drives have 50% greater media transfer rates.

Three Ultra-Wide SCSI disks saturate a single Ultra-Wide SCSI bus and adapter. Two buses support a total of six disks and a maximum read throughput of about 60 MBps. When a third Ultra-Wide SCSI adapter and three more disks were added to the system, the PCI bus limit was reached. This configuration achieved a total of 72 MBps – just over the half-power point of the PCI bus. Adding a fourth adapter showed no additional throughput, although the combined SCSI bus bandwidth of 120MBps would seem to be well within the advertised 133 MBps. While the practical limit is likely to be limited by the exact hardware configuration, the PCI half-power point appears to be a good goal.

[1] By default, *ftdisk* binds volume sets in increasing device order. This caused the first and second stripe chunks to be on the first SCSI bus and the third and fourth steps to be on the second SCSI bus.

Putting everything together, the combination of asynchronous IO, disk striping, and the NT file system are shown in Figure 10. The throughput with and without file system buffering is compared for a three-disk Ultra-Wide stripe set.

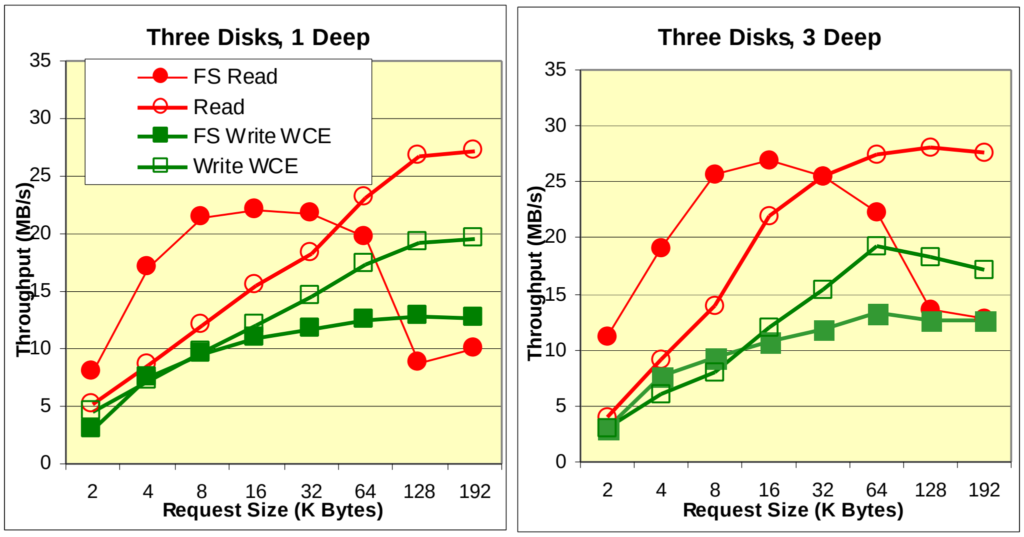


Figure 10 – Effect of file system caching – File system caching provides extra parallelism to a three-disk array by coalescing smaller writes into 64KB transfers. The graphs show the throughput of a three-drive array using unbuffered IO (WCE is enabled) and 1 to 3 deep application request depths. Read throughput is increased until 64KB.

Striping and asynchronous IO can dramatically increase performance of smaller read requests. As with the single disk results shown in Figure 5, file system prefetching increases throughput for requests smaller than 64KB. At 64KB the application request size matches the file system prefetch size. Because this is also the stripe set chunk size, each prefetch request accesses the next disk in the volume set. File system prefetching effectively distributes the load across the disks more. With 8-deep requesting, non-cached access gets better distribution with at least 8KB requests. Above 64KB request size, it is always better to avoid the file system cache copy. The half-power point is reached at smaller request sizes with file system prefetching, but large requests combined with non-buffered asynchronous IO gives better total performance and sustains at least the half-power performance above request sizes of 64KB.

Striping writes gives good scalability, but reaching the half-power point remains difficult. With 2KB or 4KB requests, write coalescing by the file system gives the same performance gain as WCE at the drive. At 8KB or above, the throughput is better with only WCE. The file system write-back is not as effective at distributing the load across the disks when caching is active within the drive. The effect of asynchronous requesting is also small due to WCE.

To summarize, issuing large and deep asynchronous requests combines to saturate a single disk. Using three-deep asynchronous requests more than doubles write throughput, even on a single disk. Using this technique, write requests in the 16KB range get performance comparable to WCE. By striping the file across 3 disks on a single controller, the bandwidth rises about 3x. The controller saturates at three disks: Fast-Wide SCSI saturates at 15MBps, Ultra-Wide saturates at about 30MBps. By adding controllers in three-disk groups to a single PCI bus, the PCI bus saturates at about 70MBps. By adding multiple PCI busses, the processor or memory bus can saturate. Figure 11 summarizes these bottlenecks. To reach the half-power point, use large transfers and either three-deep requests or WCE.

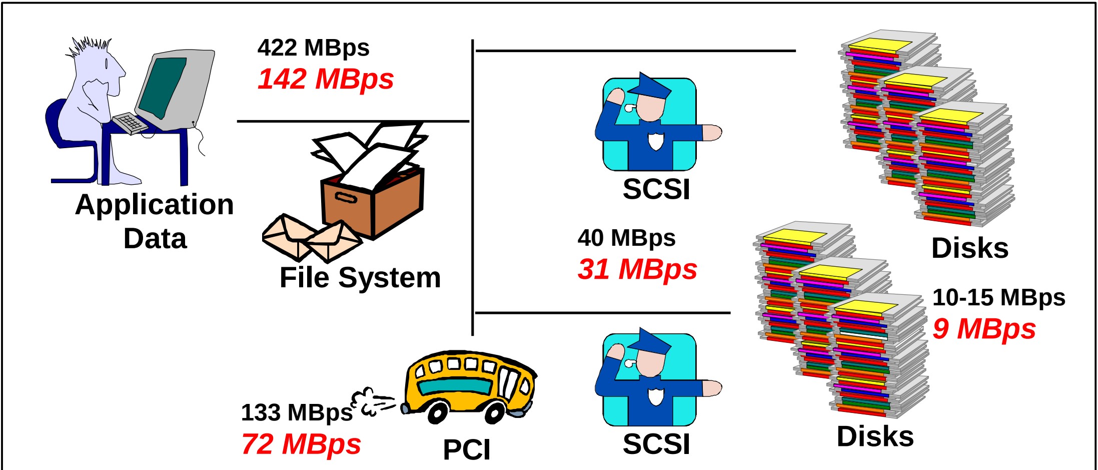


Figure 11 – Summary of Bottlenecks - PAP (peak advertised performance) vs. RAP (real application performance) – Ultra disks saturate at the media limit of 9MBps. Additional disks may be added to take advantage of request parallelism on a SCSI bus. Three drives saturate the bus and adapter. By adding busses and adapters, and three drives, throughput can grow to saturate a PCI bus. High-end systems support multiple PCI busses, and so can saturate the memory bus. Using memory resident files, our platform saturates at about 142 (read) or 93 (write) MBps due to main memory copy speeds.

5. Measurements of Device and File System Internals Performance

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 System Memory

To understand memory copy overhead associated with file system buffering, we made a number of measurements. In all cases, we moved 4MB and timed the operations using the Pentium cycle counters. Target arrays were allocated on page boundaries and we repeated the measurements varying both alignment on a page and page allocated.

We coded a number of simple data assigns and a `memcopy`(). Each data assign loop contains four `double` assigns. Floating point doubles achieved slightly better performance than integers or single precision floating point. We unrolled the loops to take advantage of the 4-deep Pentium Pro pipelining and saw a few percent gain over the tightly coded loop. We coded both true copies, moving all data in a cache line, and cache line accesses, assigning only the first `double` in a 32B line.

We also used temporary files. The NT file system attempts to hold all temporary file storage within the file cache, so accesses to these files are performed by memory copy. Temporary files are opened by including `FILE_ATTRIBUTE_TEMPORARY` when calling `CreateFile()`. Temporary file accesses are a "best case" IO performance limit for file system buffered requests with no PCI or other IO subsystem hardware bottleneck.

Table 2 summarizes the results for our test machine and a nominally identical Gateway 2000 G6-200. Both machines are 200 Mhz Pentium Pros with 64MB of 4x4 60nsec DRAM. The system memory bus is 64-bits wide and cycles at 66 Mhz (422 MBps). While both machines have identical part numbers, the machines actually differ in that the test machine has fast page mode DRAM while the "identical" machine has EDO DRAM. This difference accounts for only a difference of 10-15%. We believe the "identical" memory may have more banks as well, since greater interleaving would explain the larger variations.

Table 2 – Processor to Memory Bandwidth – Temporary file reads and writes are the "best case" limits for file system buffered IO; the data is copied from or to an application buffer but not read from or written to disk media. `Memcopy` loads the system more than the file system because it does not reuse cacheable destination buffers. While the two machines have identical product codes, the memory subsystem performance is considerably different.

Memory bandwidth (MBps)	**Test Machine (MBps)**	**"Identical" Machine** (MBps)
Unrolled DOUBLE load to single destination	88	81
Unrolled cache line read miss	164	230
Temporary file read	142	148
Unrolled DOUBLE store to single destination	47	82
Unrolled cache line write miss	50	84
Temporary file write	93	136
Memcopy (assembly code double load unroll)	47	54

Processor reads are limited both by the response of the memory subsystem and by the ability of the processor to pend requests. The factor of two difference in bandwidth between one-double-per-cash-line and read-whole-cash-line implies that the first is not memory limited – we were not able to get enough requests in flight to benefit from memory request pipelining. The interesting result is that temporary file reads are better than our unrolled `double` access, This is probably due to better (assembly tuned) coding within NT.

On our test system, write bandwidth is significantly less than read bandwidth. Writes also do not show a difference between full and partial cache line access. Since writes are asynchronous, the processor does not stall until the maximum number of pending writes has been reached. Again, temporary file writes are substantially faster than our coding.

On our “identical” system, the read and write bandwidths are comparable. We don’t believe this is experimental error , we varied the physical page layout between tests. The two systems are different.

`Memcopy` is not a good model for estimating file system buffer copy overheads. While both use hand-tuned code to move data between buffers, the memory access patterns are significantly different - `memcopy` sweeps the processor board cache generating substantially more traffic per byte moved. Temporary file writes repeatedly copy from a 64KB buffer to 4MB of file buffers; `memcopy` moves 4MB to 4MB.

The underlying details are both complex and poorly documented. The behavior depends not only on memory bandwidth, but also memory latency and cache coherency protocol. At best, the maximum delivered data rates for a pure server in which the processor does not handle the delivered data is one half the main memory bandwidth (read once, write once).

We believe that our system is primarily limited by memory and not by the memory bus. A processor cache read misses require two bus transactions: a short read request and the longer cache line read returned data. A processor cache writeback can require up to four transactions: the two transactions for a read of the line, a short intention to write, and the cache line write to memory. If the memory bus were the bottleneck, write bandwidth would be about half read bandwidth in the limit. The different results from the different machines indicate that the memory subsystem characteristics are key.

The advertised PAP of the system bus is 422 MBps. Temporary file reads achieve about 140 MBps; temporary file writes between 93 and 136 MBps. With only one PCI on the system, we observed DMA rates of 72 MBps. Whether or not these represent the half-power point of the memory, we do not know. To do better, we would have to distribute our application across processor boxes.

5.2. Disk Controller Caching and Prefetching

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

$$Request\_Service\_Time \; ? \; Fixed\_Service\_Time \; ? \; Disk\_Seek\_Time \; ? \; \frac{Transfer\_Size}{Media\_Transfer\_Rate} \; ? \; \frac{Request\_Size}{SCSI\_TransferRate}$$

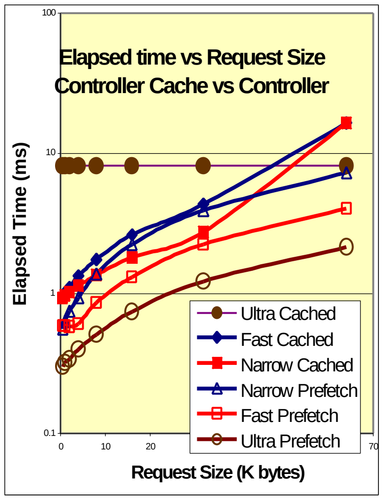
The fixed overhead term includes time for the application to issue and complete the IO, the time to arbitrate and transfer control information on the SCSI bus, converting the target logical block to physical media location. The fixed time also includes the disk controller SCSI command handling, and any 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 (>0.5ms). For large >32KB) requests, the media-transfer time (> 8ms) 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 column 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	.56	0.93	0.59	8.14	0.30
1K	1.01	.63	0.97	0.59	8.14	0.32
2K	1.11	.75	1.02	0.58	8.14	0.34
4K	1.33	.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 64KB, 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 8KB 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 [Worthington, et al]. 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.

5.3. 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.

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 4).

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%**

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.

?? **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 reSELECT 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.

5.4. 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, or 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.

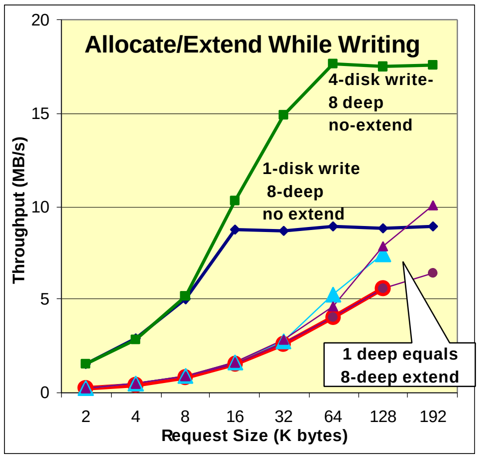


Figure 12 – File Allocate/Extend Behavior – When a file is being extended (new space allocated at the end), NT forces synchronous write behavior to prevent requests from arriving at the disk out-of-order. C2 security mandates that the value zero be returned to a reader of any byte which is allocated in a file but has not been previously 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/extend writes dramatically slow file write performance.

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 2KB 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 unbuffered asynchronous write performance, you should preallocate the file storage. If the space is not pre-allocated, the NT file system will first zero it before letting your program read it.

5.5. 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 64K 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 NT 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 512B to 4KB depending on the volume size. The stripe chunk and file system allocation unit 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.

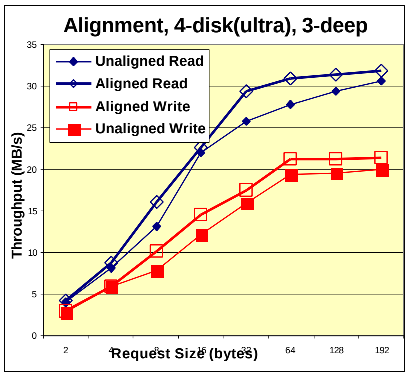


Figure 13 – Alignment Across Disks in a Stripe Set – The performance of a file aligned to the stripe chunk is compared to a file that is mis-aligned by 12KB. If requests split across stripe set step boundaries, read performance can be reduced by nearly 20% and writes by 15%. The effect is more pronounced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

Figure 13 shows the effect of this misalignment. Alignment with the stripe chunk improves performance by 15-20% at 64KB request sizes. A misaligned 64KB application request causes in two disk requests (one of 12KB and another of 52KB) 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 64KB requests active at any given time. In the misaligned case, there are 16 requests of mixed 12KB and 52KB sizes to be coordinated.

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. The format command allows allocation size to be set in increments up to 64KB. The cost of using a 64KB allocation unit is the potential of wasting disk space if the volume is used to contain many small files; the file system always rounds the file size to the allocation unit.

[2] The command is of the form '`format e: /fs:ntfs / a:64k` ' to create an NTFS 4.0 file system with 64KB allocation.

5.6. Location on the disk

Modern disks are *zoned*: outer tracks have more sectors than inner tracks and the bit rate on the outer tracks is higher. This is a natural consequence of having constant aerial density and constant angular velocity: an outer track may be 50% longer than an inner track. The bytes arrive faster and there are more bytes per seek. Figure 14 shows the throughput (MBps) when sequentially reading at different radial positions. For the Ultra-Wide drive, the PAP media transfer rate varies from 15 to 10 MBps and the synchronous RAP is about 65% of that. The PAP media-transfer rate on the Fast-Wide disk media rate varies from 8.8 MBps to 5.9 MBps; the synchronous RAP is about 75% of that. This variation across the surface is similar on other drives.

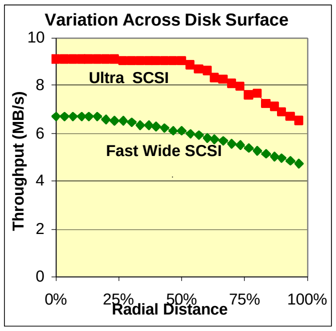
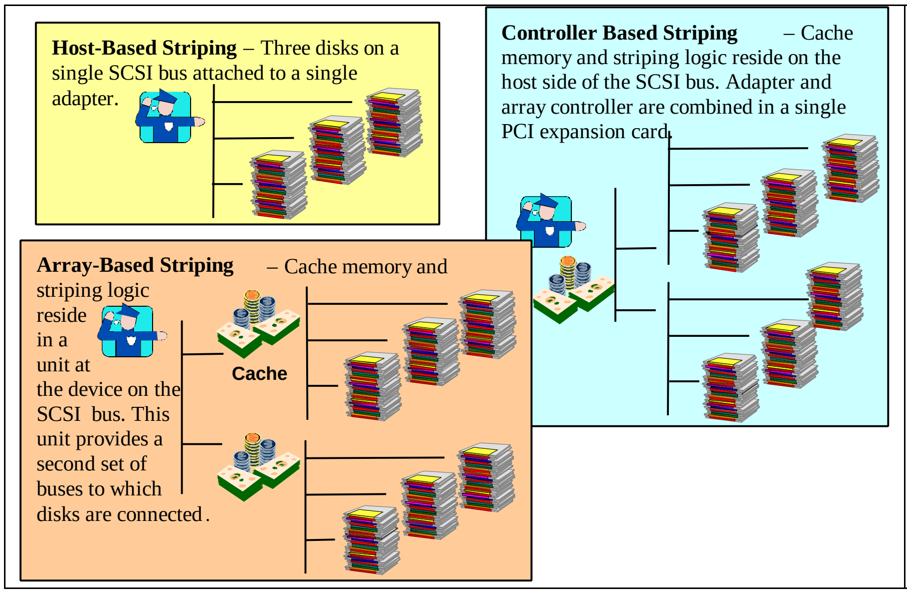


Figure 14– Variation Across Disk Surface – The media bandwidth at the inner disk tracks is up to 30% lower than the bandwidth at the outer zone. The experiment shows read rate for 64KB synchronous unbuffered reads.

File placement can help attain a disk's the half-power point. Files allocated near the outer edge of the disk will have higher throughput than files allocated near the inner zone. This also means that the throughput of a sequential scan of the entire disk decreases as it progresses across the disk surface.

5.7. Striping in Host, Host-Bus Adapter, or RAID Controller

There are several possible arrangements for doing multiple-disk striping. Figure 15 shows three possible combinations that locate cache memory in different places and provide different underlying "plumbing". These different



Host-Based Striping – Three disks on a single SCSI bus attached to a single adapter.
Controller Based Striping – Cache memory and striping logic reside on the host side of the SCSI bus. Adapter and array controller are combined in a single PCI expansion card.
Array-Based Striping – Cache memory and striping logic reside in a unit at the device on the SCSI bus. This unit provides a second set of buses to which disks are connected.
Cache

Figure 15 – Variety of Adapter/Controller Configurations – Striping can be performed in the host or within a dedicated controller. That controller can be located on the SCSI adapter or located with the disk drive array and connected through a normal SCSI adapter.

combinations exhibit significantly different properties across different workloads as shown in Figure 16. Note that the hardware striping controllers were not necessarily optimally tuned for large sequential accesses. Caching in the controller acts like file system caching to present the drive with larger requests, but without incurring the processor overhead within the host processor.

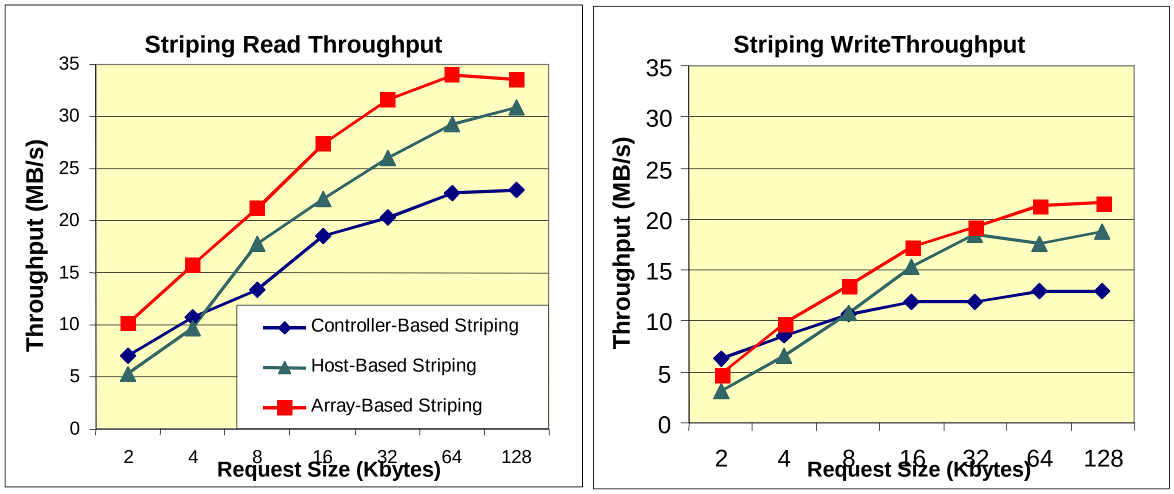
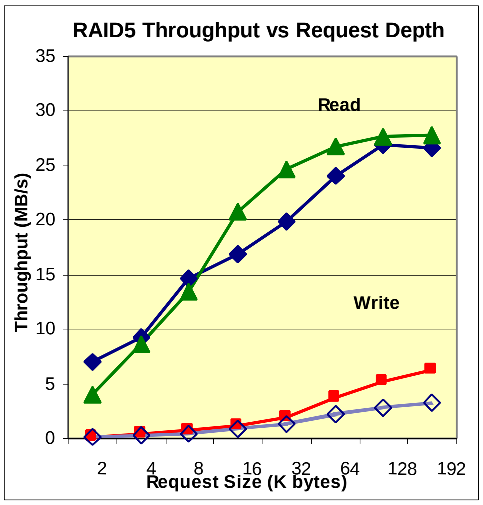


Figure 16 – Comparison of Controller-Based, Array-Based, and Host-Based Striping. The array-based solution performs significantly better. The controller-based configuration gave good performance only for small (2KB and 4KB) writes. Host-based striping gives good performance at significantly less hardware cost. Note that the array and controller solutions may not have been optimally tuned for large accesses.

5.8. RAID

In addition to simple striping, RAID 5 functionality can also be provided either on the host, or by a separate controller. The issues of host-based or controller-based RAID5 are very complex. Controller-based RAID often provides superior failover and reconstruction in case of faults. This section only considers the failure-free performance of two approaches. The purpose is just to assess the relative cost of the host-based RAID5 logic.



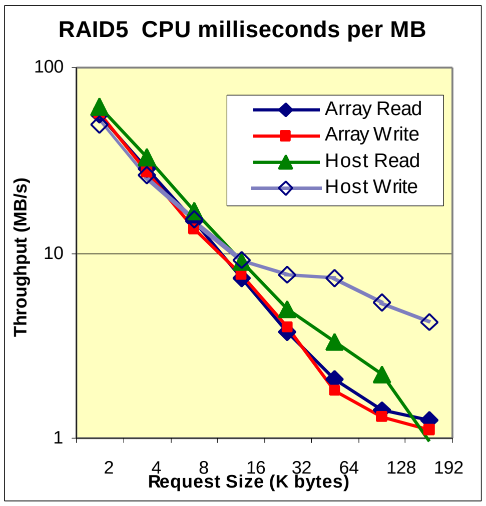


Figure 17 – Host-based vs. Controller-based RAID5 read and write. 4-Ultra disks with WCE disabled were configured as a RAID5 stripe set both using host-based NT fault-tolerant RAID and using a RAID5 array. The read performance of the NT file system compares to the controller's performance. For writes, host-based solution is competitive on small requests, but for requests of 32KB and beyond, the host-based solution consumes more processor doing software checksums and consequently has lower overall throughput.

Figure 17 compares the performance of a four-disk RAID5 stripe set using host-based striping and a hardware disk array. We see that the array performs slightly better on writes, while reads are about equal, with host-based performing somewhat better at medium request sizes. More importantly, the chart on the right shows the impact on the host processor of the two options. Using the disk array to manage RAID5 allows the parity calculations on writes to be offloaded from the host processor, greatly reducing the cost per write requests of 32KB and beyond. We would see a similar effect for reads if we caused on of the disks to fail and reach read request had to reconstruct the original data from the parity.

6. Summary and Conclusions

The NT 4.0 file system out-of-the-box sequential IO performance for 4KB requests is good: reads are close to the media limit, writes are near the half-power point. This good performance comes at some cost; the file system is copying every byte, and coalescing disk requests into 64KB units. Write throughput can be nearly doubled by enabling WCE, although using WCE does risk volume corruption if the disk cache is lost. Another alternative is to use large requests and make three-deep asynchronous requests.

NT file striping across multiple disks is an excellent way to increase throughput. The file striping must be combined with large and deep asynchronous requests to keep enough data in the pipeline for the disks. Data is striped in 64KB chunks, so one needs approximately *N+1* outstanding sequential IOs to keep *N* drives busy. That is 250KB of outstanding IO to drive an array of 3 disks at speed.

An application can saturate a SCSI bus with three drives. By using multiple SCSI busses (and disks), the application can saturate a PCI bus. By using multiple PCI buses, the application could saturate the processor bus and memory subsystem. With current (mid 1997) Intel platforms, the processor can access temporary files at about 140 MBps. All this was summarized in Figure 11. For unbuffered IO, these processors and the software is capable of driving 480 MBps.

If the system configuration is balanced (disks do not saturate busses, busses do not saturate one another), the NT file system can be programmed to reach the half-power point. Indeed, 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) file striping, and (5) unbuffered IO.

Write performance is often worse than read performance. The main pitfalls in writing files are: (1) If a file is not already allocated, NT 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, then 2KB buffered writes will first read a 4KB unit and then overwrite. (3) If the RAID chunk size is not aligned with the file system allocation size, the misalignment causes large requests to be broken into two smaller requests split across two drives. This doubles the number of requests to the drive array.

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

?? Larger requests are faster. Requests should be at least 8KB, 64KB if possible.

?? Small requests consume significantly more processor time per byte than larger ones. Doing 2KB sequential IO requests consumes more than 30% of the processor. Using 64KB requests goes faster and consumes less than 3% of the processor.

?? If you absolutely must make small requests, double buffering is not enough parallelism. There are noticeable gains through 8-deep requests;

?? Write-Cache-Enable at disk drives provides significant benefits for small requests. Issuing three-deep asynchronous requests comes close to WCE performance for larger requests. WCE risks data loss and/or volume corruption; asynchronous requests do not.

?? 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 64KB disk requests for both reads and writes. This provides significant performance improvement for requests smaller than 64KB.

?? At 64KB and larger requests, file system buffering degrades performance significantly 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 if requests are issued asynchronously. Consequently, files should be truncated before they are re-written or they should be re-written in multiples of 4KB.

?? Array controllers improve performance by varying the location of the caching and read-ahead logic. The benefit varies with workload.

This paper gave a basic tour of the parameters that affect sequential I/O performance in NT. Many areas are not discussed here and merit further attention. As discussed in Section 5, programs using asynchronous I/O have several options for managing asynchronous requests, including completion routines, events, completion ports, and multi-threading. The benchmark used completion routines in an otherwise single-threaded program. We have not explored the tradeoffs of using the other methods.

The 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. The impact of competing application streams is an active research topic.

The RAID5 discussion was superficial. It ignored failure and reconstruction behavior. There are an enormous number of options in configuring and tuning RAID systems that we have not explored. Particular applications may use of different RAID levels, different striping parameters, and a greater variety of hardware options. We have only shown the simplest comparisons to give a basic idea of the variety available to system designs.

8. Acknowledgements

Tom Barclay did the initial development of `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 Adaptec 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, 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 helped in understanding NT internals.

9. References

[Custer1] Helen Custer, *Inside the WindowsNT™ File System*, ISBN 1-55615-481, Microsoft Press, Redmond, WA. 1992.

[Custer2] Helen Custer, *Inside the WindowsNT™ File System*, ISBN 1-55615-660, Microsoft Press, Redmond, WA. 1994.

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

[Richter] 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, Redmond, WA. 1995.

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

[SCSI] *ANSI X3T9.2 Rev 10L, 7-SEP-93*. aka: the SCSI-2 Spec. American National Standards Institute (now called NIST), http://www.ansi.org.

[Worthington] 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.

Appendix: Benchmark program structure

NT supports several mechanisms for asynchronous IO. The benchmark application used in this study uses `ReadFileEx()`, `WriteFileEx()`, and IO completion routines. The code is very similar to the sample in Figure 21. That program keeps eight asynchronous requests outstanding at any given time. When an IO completes, the completion routine simply issues another request into the same buffer until the end of file is reached.

Rather than start with that complicated program, we first show the simpler routines to do synchronous reads, unbuffered IO, and event-based asynchronous reads. Richter's book has an excellent tutorial on these issues [Richter].

The simplest code to sequentially read a file is shown in Figure 18. Code that includes error handling is on the web site at http://www.research.microsoft.com/barc/Sequential_IO/. The code in Figure 18 issues a series of `64KB` requests and refills the single buffer, `cBuffer`, until reaching the end of file. The file system prefetches and buffers all data. When requests arrive, they are serviced by copying the data from the file system cache. All application requests are synchronous. The `ReadFile()` call does not return until all the necessary data has been read from disk into the file system cache and then copied to the application buffer. File system and disk controller prefetching creates pipeline parallelism. If the application request size were smaller, the file system would coalesce the sequential requests into 64KB requests to the disk subsystem. The `FILE_FLAG_SEQUENTIAL_SCAN` tells the file system that it can aggressively age these pages from the buffer pool.

```
#include <stdio.h>
#include <windows.h>

int main()
{       const int iREQUEST_SIZE = 65536;
        char cRequest[iREQUEST_SIZE];
        unsigned long ibytes;

        HANDLE hFile = CreateFile("C:\\input.dat",           // name
                                   GENERIC_READ,             // desired access
                                   0,                        // share mode (none)
                                   NULL,                     // security attributes
                                   OPEN_EXISTING,            // pre-existing file
                                   FILE_ATTRIBUTE_NORMAL     // flags & attributes
                                   | FILE_FLAG_SEQUENTIAL_SCAN,
                                   NULL);                    // file template

        while(ReadFile(hFile,cRequest,iREQUEST_SIZE,&ibytes,NULL) ) // do the read
        {       if (ibytes == 0) break;                      // break on end of file
                /* do something with the data */ };

        CloseHandle(hFile);
        return 0;
        }
```

Figure 18 –Basic Sequential Read Code: Read all of a file synchronously using the file system and a single 64KB request buffer. To optimize disk request coalescing and minimize cache pollution, the program requests the `FILE_FLAG_SEQUENTIAL_SCAN` attribute. To bypass the file system cache entirely and save the extra memory copy, the program might add the `FILE_FLAG_NO_BUFFERING` (in which case the `cRequest` would have to be aligned to the disk sector size.)

The simplest code to do unbuffered sequential file writes is shown in Figure 19. Code that includes error handling is on the web site at http://www.research.microsoft.com/barc/Sequential_IO/. `VirtualAlloc()` creates storage that is page aligned and so is disk sector aligned.

```
#include <stdio.h>
#include <windows.h>

int main()
{      const int  iREQUEST_SIZE = 65536;
       LPVOID     lpRequest = VirtualAlloc(NULL,iREQUEST_SIZE,MEM_COMMIT,PAGE_READWRITE);
       unsigned longibytes;
       int i=0;

       DWORD FileFlags = FILE_ATTRIBUTE_NORMAL | FILE_FLAG_SEQUENTIAL_SCAN |
                         FILE_FLAG_NO_BUFFERING;

       HANDLE hFile = CreateFile("C:\\output.dat",   // name
                                 GENERIC_WRITE,              // desired access
                                 0,                    // share mode (none)
                                 NULL,                 // security attributes
                                 OPEN_ALWAYS,          // create disposition
                                 FileFlags,            // unbuffered access requested
                                 NULL);                // file template

       do  // write 100 64KB blocks
           {/* fill buffer with data */
           WriteFile(hFile, lpRequest, iREQUEST_SIZE, &ibytes, NULL);
           } while(i++ < 100);

       CloseHandle(hFile);
       VirtualFree(lpRequest, iREQUEST_SIZE, MEM_COMMIT);
       return 0;
       }
```

Figure 19 –Basic Sequential Unbuffered Write Code: This program synchronously writes ten 64KB blocks to a file. Opening the file with the `FILE_FLAG_NO_BUFFERING` flag requests that buffering be suppressed. This bypasses the file cache entirely and so avoids polluting it and saves the extra memory copy. All application write requests are synchronous. The `WriteFile()` call does not return until all the necessary data has been written to disk. Unbuffered

```
#include <stdio.h>
#include <windows.h>
#include <winerror.h>

// 64-bit move – allows compiler to optimize coding
#define Move64(Destination, Source) \
   *((PULONGLONG) &( Destination)) = *((PULONGLONG) &( Source))

int main()
{   const int iDEPTH = 8;
    const int iREQUEST_SIZE = 1 << 16;
    unsigned long ibytes;
    DWORD FileFlags = FILE_ATTRIBUTE_NORMAL |
                      FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED;
    ULONGLONG HighWaterMark = 0;
    struct {                           // array of request buffers and overlap structs
           LPVOID     lpRequest;
           OVERLAPPED Overlap;
           } Req[iDEPTH];

    for (int i = 0; i < iDEPTH; i++) {
        memset(&Req[i].Overlap, 0, sizeof(Req[i].Overlap));
        Req[i].Overlap.hEvent = CreateEvent(NULL,FALSE,FALSE,FALSE);
        Req[i].lpRequest= VirtualAlloc(NULL, iREQUEST_SIZE, MEM_COMMIT, PAGE_READWRITE);}

    HANDLE hFile = CreateFile("C:\\output.dat",      // name
                              GENERIC_WRITE,         // desired access
                               0,                    // share mode (none)
                               NULL,                 // security attributes
                               OPEN_EXISTING,        // preallocated file
                               FileFlags,            // overlapped async
                               NULL);                // file template

    for (i = 0; i < 100; i++){         // Do 100 writes of 64KB each
        int j = i % iDEPTH;                  // j is the request modulus (0…8
        if (i >= iDEPTH)                     // if past startup stage, wait for previous
           {WaitForSingleObject(Req[j].Overlap.hEvent, INFINITE);}  // request to complete
        memset(Req[j]. lpRequest , i, iREQUEST_SIZE); // use the buffer, eg. Fill it
        Move64(Req[j].Overlap.Offset, HighWaterMark); // set file offset to write
        WriteFile(hFile, Req[j]. lpRequest , iREQUEST_SIZE, &ibytes, &Req[j].Overlap);
        HighWaterMark += iREQUEST_SIZE;              //
        }

    CloseHandle(hFile);                // should also free buffers and event handles
    return 0;
    }
```

```
#include <windows.h>
#include <winerror.h>
// 64-bit increment
#define IncrementOffset64(Offset,Increment)  *((PULONGLONG) &(Offset)) += (LONGLONG) (Increment)
//-------------------GLOBALS------------------------------------------------------------
const int      iDEPTH         = 8;           // max depth of IO pipeline
const int      iREQUEST_SIZE  = 1 << 16;     // request size is 64KB
HANDLE         hFile          = NULL;        // the file
HANDLE         hDoneEvent     = NULL;        // mainline waits for this event once pipe is full
int            iOutstandingIOs = 0;          // how many IOs are in progress?
LPVOID         lpData[iDEPTH];               // Pointers to request buffers
OVERLAPPED     Overlap[iDEPTH];              // Overlap structure for each buffer
//-------------------HELPER ROUTINE-----------------------------------------------------
VOID WINAPI IoCompletionRoutine(             // Complete an asynch I/O
                               DWORD dwError,                // I/O completion status
                               DWORD dwTransferred,          // Bytes read/written
                               LPOVERLAPPED pOverlap)        // Overlapped I/O structure
       {
       int i =  (int) pOverlap->hEvent;      // index is in unused (overloaded) event cell
       //-------------------------------------------------------------------------------
       // If =not at end-of-file, issue the next IO.
       // Note that by issuing an IO from within the routine, we ris k recursion.
       // This may overflow the stack. The paper's benchmark program had this feature.
       // This is a common error. It appears in several online and reference examples.
       // A better design posts an event for the parent routine and lets th e parent issue
       // the next IO.

       IncrementOffset64(Overlap[i].Offset,(iREQUEST_SIZE*iDEPTH)); // set next address
       if (ReadFileEx(hFile, lpData[i], iREQUEST_SIZE ,&Overlap[i],IoCompletionRoutine))
              return;
       // Else,if at EOF or error, start sh utdown.
       iOutstandingIOs -=1;                                 // decrement count of IOs
       if (iOutstandingIOs == 0)                            // if all complete
              SetEvent(hDoneEvent);                         // signal main line waiter
       return;                                              //
       };                           //-----------END OF HELPER ROUTINE --------------

//-------------------MAIN ROUTINE-------------------------------------------------------
void main (void){
    hDoneEvent = CreateEvent(NULL, FALSE, FALSE, NULL);     // event parent waits on

    DWORD FileFlags = FILE_FLAG_SEQUENTIAL_SCAN | FILE_FLAG_NO_BUFFERING |  FILE_FLAG_OVERLAPPED;
    hFile = CreateFile("C:\\output.dat",                    // name
                       GENERIC_READ,                        // desired access
                       0,                                   // share mode (none)
                       NULL,                                // security attributes
                       OPEN_ALWAYS,                         // create if not already there
                       FileFlags,                           // flags & attributes
                       NULL);                               // file template

    for (int i = 0; i < iDEPTH; i++) {                      // launch an IO for each request buffer
              memset(&Overlap[i], 0, sizeof(Overlap[i]));   //  zero the structure
              Overlap[i].hEvent = (void *) i;       //readx, writex ignores hEvent, overload it.
              lpData[i] = VirtualAlloc(NULL, iREQUEST_SIZE, MEM_COMMIT,PAGE_READWRITE);
              IncrementOffset64(Overlap[i].Offset,(iREQUEST_SIZE*i)); // set next address
              if (! ReadFileEx(hFile,lpData[i], iREQUEST_SIZE, &Overlap[i],IoCompletionRoutine))
                     break;                                 // break if read past eof
              iOutstandingIOs +=1;                          //
              }

    WaitForSingleObjectEx(hDoneEvent,TRUE, INFINITE);       // wait for all IO to complete

    CloseHandle(hFile);                                     // also free events and buffers
    return;
```

There are two disadvantages to completion routines. First, there is only one execution thread. The completion routine executes only when the primary thread waits. This is not a concern for the uni-processor system used in this study, but the benchmark will not take best advantage of additional processors on a multi-processor system. Second, invoking some system calls from within the completion routine can cause stack overruns. In particular, invoking `ReadFileEx()` from within the completion routine can cause the completion routine to be re-invoked whenever other previous IO request have completed. This intermittent load-related failure can occur with large buffer depths and relatively short IO requests. Alternative better design uses either events or IO completion ports. On a multi-processor, it may be appropriate to use multiple threads.

Deep application buffering with very large buffer sizes can also cause system-tuning problems. During the time that an IO request is pending, the system locks down the physical memory comprising the buffer. This can cause memory pressure on other applications. In the extreme, IO requests will fail with reported error of `WORKING_SET_QUOTA_EXCEEDED`. The failure is not actually a process working set limitation. Rather it indicates that the memory allocated to the kernel and locked down for in-progress IO requests has exceeded an NT threshold.

The online samples at the web site contain extended programs that include error handling.