Windows Hardware and Driver Central

Disk Subsystem Performance Analysis for Windows

March 2004

Abstract

Analyzing storage subsystem performance is an art, not a science. Each rule has an exception; each system designer or administrator has a different combination of hardware configurations and software workloads to consider. This paper examines the performance of storage subsystems used by computers running the Microsoft® Windows® 2000, Windows XP, or Windows Server™ 2003 family of operating systems.

This paper considers performance from both the hardware and software perspectives. In addition, it discusses tools for storage subsystem analysis and design and provides rules of thumb and guidelines for system design and to solve the performance bottlenecks in specific configurations.

Contents

Disclaimer

Introduction

Analyzing storage subsystem performance is an art, not a science. Every rule has an exception; every system designer or system administrator has a different combination of hardware configurations and software workloads to accommodate. To fully understand the behavior of storage subsystems, one must understand every component along the path, from the user applications to the file systems to the storage driver stack to the adapters, controllers, buses, caches, and hard disks. Few people have the time and resources to accomplish this heroic task, so one is left with rules of thumb and general guidelines that inevitably need tweaking to maximize storage performance on any given system.

Decisions about how to design or configure storage software and hardware almost always take performance into account. Performance is always sacrificed or enhanced as the result of trade-offs with other factors such as cost, reliability, availability, or ease of use. Trade-offs are made all along the way between application and disk media. Application calls are translated by file cache management, file system architecture, and volume management into individual storage access requests. These requests traverse the storage driver stack and generate streams of commands presented to the disk storage subsystem. The sequence and quantity of calls as well as the subsequent translation can enhance or degrade performance.

The layered driver model in Windows sacrifices a bit of performance for maintainability and ease of use (in terms of incorporating drivers of varying types into the stack). I/O hardware paths vary widely in terms of width, speed, fan-out, complexity, space, power, and so on. Storage peripherals range from single-spindle hard disks to massive, power-protected, data-redundant cabinets full of hard disks holding terabytes of data and gigabytes of cache. Every step along the software and hardware path involves trade-offs in design and configuration.

Perhaps the most difficult part of disk storage analysis is the tendency of each component to disguise the true nature of what lies behind it. The obvious example of this is a hardware array controller that presents to the operating system the illusion of a number of single-spindle "disks" when in fact one or more of the disks might be backed by multiple or even shared hard disks that might not have redundancy built into their configuration.

This paper provides some insight into the performance of storage subsystems used by computer systems running Windows. It examines performance from both hardware and software perspectives. Tools and rules of thumb are discussed and guidelines are suggested to help system designers and system administrators better solve the performance bottlenecks in specific configurations.

This paper addresses the following topics:

- Definitions and Terminology
- Application Characteristics and Considerations
- Windows-related Characteristics and Considerations
- Hardware, Firmware, and Miniport Driver Characteristics and Considerations
- Tools for Storage Performance Analysis
- Rules of Thumb
- Summary and Future Work
- References

Definitions and Terminology

This section provides definitions of terms as used in this paper.

Disk-related Terms

Spindle

Generally used to indicate a single set of rotating platters with a ganged actuator—that is, what the average person thinks of when hearing the words "hard disk." Although this may not be a familiar term, it has the advantage of being unsullied by any association with volumes, partitions, and so on. To be technically accurate, the spindle is really just the central post that spins the media platters attached to it.

Disk (generic term)

Used to indicate an entity that appears to be a single spindle (at some specific level of the path), but could in fact be something entirely different (such as a disk array).

Physical disk

Used as in the Perfmon counter set with the same name. Hardware storage controllers (for example, SCSI controllers on a PCI bus, on-board array controllers, array controllers off some Fibre Channel fabric, or a simple IDE controller on a SouthBridge chip) present the operating system with what appears to be a set of individual spindles. Each such physical disk can be divided by the partition manager (Partmgr.sys) into partitions that are then presented to the volume manager.

Logical disk

Used as in the Perfmon counter set with the same name. One or more partitions are presented to a file system by a volume manager (for example, Ftdisk.sys or Dmio.sys) as a single logical disk, or volume. In the case of raw I/O, the disk sectors are managed by the application directly, rather than using a formal file system. For example, Microsoft SQL Server™ can be configured to achieve a performance boost by taking on this management activity and using the raw file system, which has a shorter code path than a normal file system.

If multiple partitions are combined in a single logical disk, then various disk array techniques can be used to make trade-offs between performance, reliability, availability, and capacity/cost. Such an array is termed a software-managed array, because the operating system has knowledge of the individual array components and is in charge of handling any special operations, such as read-modify-write operations on a RAID-5 array. That is not to say that the operating system has full knowledge of the entire path out to the spindles containing data for this array, but rather it has full knowledge and management of this particular array's activity at this level.

Array controller, LUN

Array controller usually refers to a hardware controller specifically designed to use one or more disk array techniques to make trade-offs between performance, reliability, availability, and capacity/cost. Such an array is termed a hardware-managed array, because the controller hides knowledge of the individual array components from the operating system and is in charge of handling any special operations—again, like read-modify-write operations on a RAID-5 array.

In the case of large subsystems with multiple hardware arrays, the individual arrays are sometimes referred to as LUNs (Logical UNits).

Logical block, LBN

Logical block refers to a specific offset on a disk, and is referenced by its logical block number (LBN). So a disk request might specify that 8 logical blocks be read from a disk starting at logical block number 328573. In almost all systems, logical blocks are equivalent to sectors, which are currently standardized at 512 bytes. That being said, it is possible to change the sector size on a hard disk, and operating systems can and will use other sizes—especially as spindle capacities increase. In this paper, logical blocks and sector sizes are assumed to be 512 bytes.

Array-related Terms

There are academic reasons to avoid the use of the RAID (Redundant Array of Inexpensive/ Independent Disks) terminology, but it has become somewhat standardized by the industry and so it will be used in this paper—but not exclusively. The pros and cons of each strategy will be discussed later in this paper.

JBOD (pronounced “jay-baud”)

Just a Bunch Of Disks. This is the opposite of an array; individual disks are referenced separately, not as a combined entity. Of course, what appears to be a JBOD set of disks at one level might be arrays of spindles at another level.

RAID-0

Originally known as striping, this strategy parcels out logical blocks round-robin to each disk in the array. Typically each disk receives multiple contiguous LBNs in each chunk, or **stripe unit**. The total size of these contiguous chunks passed out to each disk is called the **stripe unit size**. The stripe unit size multiplied by the number of disks in the array (excluding any parity units) is the **stripe size**. So, for example, if each disk receives 64K of consecutive LBNs at a time and there are 8 disks worth of data in the array, then a stripe contains 512K—one stripe unit from each disk containing data.

RAID-1

Originally known as mirroring, duplexing, or shadowing, this strategy keeps an exact duplicate of data from one portion of a given disk on an equivalent-sized portion of another disk. In this simplest case, two identically-sized disks are kept as perfect duplicates of each other—any write requests are “mirrored” to both disks.

RAID-0+1

Also known as RAID 1+0 or RAID 10, this is a combination of RAID-0 and RAID-1 techniques, where data is striped across multiple disks and every piece of data has a mirrored copy on another disk. Mirroring striped arrays is probably the most common combination of techniques, but striping mirrored sets might be the best choice given current array controller heuristics (see “Redundancy through Replication” later in this paper).

RAID-5

Also known as striped parity or rotated parity, this adds redundancy to the striping strategy by adding another spindle to an array. This additional capacity provides storage for an XOR checksum across all of the disks in the array. The checksum provides parity data that is stored throughout the array, so that all disks have both data and some interspersed parity information.

There are other RAID schemes, but the above strategies are the most common. In some cases, companies have invented new RAID numbers for marketing reasons. For example, the addition of large or specialized caches to a hardware array unit have resulted in larger-numbered RAID labels without any change to the array data layout or management strategy.

Reliability and Availability Terms

Reliability

Refers to issues of data loss or corruption. A storage subsystem is reliable if a read request eventually returns (with very high degree of probability) the same data sent to the media by the most previous write request to the same LBN (or requests to the same LUNs).

Availability

Refers to timeliness in accessing data. A highly-available storage subsystem continues to offer data access even if there are multiple hardware failures along the path (or paths) between host and spindles.

Hot spare

An empty disk kept available for immediate inclusion into a failed array. That is, it can quickly take on the responsibility of a failed mirror disk or a failed disk in a RAID-5 array. The data that was on the failed disk is regenerated or reconstructed from the remaining disks and placed on the hot spare, after which array performance and reliability resume normal levels.

Application Characteristics and Considerations

Understanding application behavior is essential for both storage subsystem planning and performance analysis. The better the workloads on the system are understood, the more accurate the planning and analysis.

If possible, the system’s aggregate workload should be divided into the individual contributors—applications, operating system background, backup requests, and so on. Often they are directed at different pieces of the storage subsystem, which makes them easier to separate. Each workload can be characterized using the following axes of freedom:

- Read:write ratio
- Sequential/random (some measure of temporal and spatial locality)
- Request sizes
- Interarrival rates, burstiness, concurrency (interarrival patterns)

Each workload will most likely be composed of a mixture of sub-workloads—for example, sequential 8K reads mixed with random 4K writes. This paper discusses a number of tools available to help separate and characterize workloads.

Ideally, the aggregate system workload can be reduced to a set of individual components, such as:

- Random 4K reads to the main database files, fairly constant during business hours, 2000 requests per second
- Random 16K writes to the main database files, in bursts of 50–100 requests, 10 requests per second (on average)
- Sequential 8K writes to the database log file, fairly constant during business hours, 100 requests per second
- Sequential 64K writes to the backup subsystem, constant from midnight to 1:00 a.m., 1000 requests per second
- Sequential 64K reads from the main database files, constant from midnight to 1:00 a.m., 1000 requests per second

Such data combined with reliability, availability, and cost constraints can be used to analyze an existing system or design a new one capable of meeting specific criteria.

This data can be refined even further if more detailed analysis is required. For example, the individual distributions of locality, request size, and interarrival rates can be tracked for each workload, rather than just mean values. However, the correlations between the various characteristics compound the difficulty of the analysis, so the typical system designer or administrator should focus on first-order effects.

If workload characteristics are known in advance, whether obtained from empirical data or detailed modeling, an understood hardware configuration can be evaluated as to how it will perform under that workload. This section contains many guidelines to aid in understanding the trade-offs made in designing and configuring a storage subsystem to service a given workload while meeting specific performance, reliability, availability, and cost criteria.

Simple Workload Characteristic Considerations

Read requests can result in prefetching activity at one or more points along the storage path. In the case of sequential read workloads, this will usually provide a performance advantage. However, for random read workloads, any prefetching activity is wasted and might in fact interfere with useful work by delaying subsequent request service or polluting any caches at or below the prefetcher.

Write requests can be combined or coalesced if a buffer or cache along the way can collect multiple sequential writes and if the cache controller can issue a large write request covering all of the write data. Completion status is returned in the same manner as if all of the writes completed at the same time. That is, if the individual writes at the coalescing point would normally have immediately responded with completions to the issuing entities from that point, then they will still do so. If they would have waited for completion confirmation from farther along the path before responding, then they will wait until the completion confirmation comes for the large (combined) write request. The typical scenario is a battery-backed controller cache providing immediate completion responses and allowing long streams of potentially-serialized small writes to be coalesced.

Caches are used to improve performance for both spatial and temporal locality. Prefetching into a read cache is an example of a locality optimization; it assumes that if LBN N is read, then LBN N+1 will probably be read soon. Write-back caches provide a similar optimization, assuming that dirty data will either be overwritten (temporal locality) or can be coalesced as adjacent LBNs are dirtied (spatial locality). Write-back caches also allow disk requests to be reordered (or scheduled). Disk schedulers are usually designed to reduce disk request service time, but other criteria might be added to the scheduling heuristic, such as starvation avoidance or prioritization of specific request types.

Request size is of particular significance in striped arrays and is covered in more detail in “Stripe Unit Size” later in this paper. In some cases, Windows might divide disk requests into 64K chunks.

Few systems experience a steady-state workload where the characteristics of the requests do not fluctuate over time. Most systems experience bursts of activity and idle periods, often tied to the 24/7 cycles of the work day/week. For example, the activities performed online at a local bank can differ dramatically as the day progresses: the bank staff first arrives in the morning, the bank opens its doors, the lunch hour, the bank closes its doors, the bank staff leaves, and finally automated reporting and backup start up. In addition, there are differences in workload throughout the work week (for example, paydays) and into the weekend. Other cycles might be quicker, perhaps when a logger periodically flushes its buffers to disk or when memory pressure causes paging to occur until the pressure is released. Characterizing a workload's peaks is perhaps even more important than understanding its average or steady-state components.

Similarly, the workload distribution across the various disks in the storage subsystem may also change over time, especially if different workload components use different disks. For example, a backup application might have a dedicated set of disks or other storage devices.

Data Layout and Redundancy Considerations

The characteristics of a workload—together with the performance, reliability, availability, and cost criteria—provide the necessary ingredients to begin designing a workable storage subsystem.

JBOD

Single-spindle hard disks will function as the baseline against which to consider other data layout and redundancy schemes. Spanned disks—which basically append one spindle's LBNs onto another without any real change in data layout or redundancy—have the same general pros and cons as single-spindle JBOD.

If logical disks are each on different spindles (or spanned spindles), there is a data isolation advantage. If a single disk dies in a single-spindle JBOD configuration, only the logical disk for that spindle is gone; all remaining logical disks can continue to serve data requests.

For workloads consisting entirely of small random requests, no load-balancing schemes are going to be of any use in improving performance (over JBOD).

For workloads with some degree of sequentiality or locality of reference, load balancing must be considered. Concurrency becomes a major factor; if each logical disk is servicing a separate stream of sequential activity, JBOD will provide better performance than a striped configuration where seek times would rise dramatically as each spindle seeks to multiple regions in its LBN space. If, however, the number of streams is small, then JBOD will have some disks idle while others might be 100% utilized.

JBOD scorecard (baseline):

- Performance:
  Susceptible to static and dynamic "hot spots" (load imbalance). Works well for workloads of small random requests or concurrent streams of sequential requests to multiple volumes.
- Reliability:
  Provides data isolation between logical disks; each spindle failure only affects the data on the logical disk related to it.
- Availability:
  Provides data isolation between logical disks; each spindle failure only affects access to the logical disk contained on it.
- Capacity/Cost:
  Baseline.

Disk Array Trade-offs

For all levels of server systems, disk arrays have replaced JBOD as the preferred means of disk storage configuration. This section examines how workload characteristics directly factor into the design or behavior of a storage subsystem.

The term RAID was coined by Dave Patterson's research group around 1990. However, there are old patents from IBM and DEC that cover most of what exists in modern arrays. For an overview of the various array permutations, see [Ganger94]. Basic array descriptions can be found at numerous websites, but anything from a company that actually builds array products might be heavily biased.

After the discussion in this section of each array type, a summary is provided of its performance, reliability, availability, and capacity/cost characteristics. For the equivalent JBOD summary, see the previous section.

Striping (RAID-0, RAID-0+1, RAID-5)

This paper discusses coarse-grain striping, where the stripe unit size is a multiple of the disk sector size. When stripe unit sizes are smaller than a sector (for example, a byte or a bit), multiple disks are required to service any request. Such fine-grain striping is useful in high-throughput streaming environments where N spindles can be grouped to function as a single disk with N times the throughput.

Coarse-grain striping allows large requests to engage multiple disks for higher throughput, but at the same time it allows the array to concurrently service multiple small requests. Assuming a stripe unit size is chosen to match the workload, this form of striping provides automatic load-balancing. Small requests are distributed across all disks, with the round-robin placement policy acting as a hashing function. This prevents the formation of static or dynamic "hot spots," that is, small ranges of LBNs receiving intense bursts of requests. Load-balancing for large requests is a non-issue, as all (or most) disks will participate in servicing the requests.

The most important factor about coarse-grain striping is the stripe unit size. Large stripe units enhance concurrency by allowing multiple requests to be serviced independently, reducing overall positioning delays. This can reduce queuing delays and increase throughput. Small stripe units spread requests across multiple disks, reducing data transfer times for individual requests. For more information, see "Stripe Unit Size" later in this paper.

Striping scorecard (as compared to JBOD):

- Performance:
  With an appropriately chosen stripe unit size, striping can provide load-balancing, protect against hot spots, reduce response times, increase throughput, and increase request concurrency. A poorly chosen stripe unit size has the opposite effect—increasing response times or reducing throughput by reducing request concurrency, allowing the formation of dynamic or static hot spots, or failing to take advantage of locality in the workload.
- Reliability:
  As file data is striped across all disks, losing a single disk basically corrupts all N disks worth of data. In a JBOD scheme, losing a single disk might only destroy 1/Nth of the total data—for example, if the disks have separate file systems or are managed as completely independent raw devices.
- Availability:
  If connectivity is lost to one disk, it is extremely unlikely that much can be done using just the remaining N–1 disks. In a JBOD scheme, losing a single disk

might only affect a portion of the workload, perhaps allowing some applications to proceed normally.

- Capacity/Cost:
  Equivalent to JBOD.

Redundancy through Replication (RAID-1, RAID-0+1)

For numerous reasons, including the inability of magnetic tape to keep up with magnetic disk and the decrease in Mean Time Before Failure (MTBF; that is, the probability of failure increases) as the number of disks in an array increases, it has become impractical not to have some form of online reliability safety net. Perhaps the simplest form is keeping two or more copies of mission critical data on separate spindles, subsystems, or perhaps even in different geographical locations. If a disk fails, all of the data is still available on its mirror disk.

Non-redundant striped arrays are especially vulnerable to data loss, as one disk failure destroys the entire array's data. Medium- and large-sized files are divided into stripe-unit-sized chunks and distributed across the array. Every file with a chunk on a deceased disk is no longer available. This is why striping is often combined with mirroring. In a RAID-0+1 configuration, the increased danger of striping data across multiple disks is mediated by the fact that all single-disk and many multi-disk failures are survivable; only multi-disk failures that affect all of the disks in a specific mirrored set result in loss of the array's data. For example, in an array of 10*2 mirrored disks, there are 190 different combinations of 2-disk failures, and only 10 of those possibilities would result in data loss.

There is some debate about RAID-0+1 versus RAID 1+0, but this is a controller design issue, not a data layout issue; the resulting data layout is **identical** for both approaches. If striped arrays are mirrored, it is likely that a single disk failure will cause a controller to take an entire striped array out of action, thereby increasing the susceptibility of the overall system to an additional disk failure. If mirrored arrays are striped, a single disk failure will generally not cause a controller to remove a mirrored set, but will activate some sort of operator warning mechanism or automatically rebuild the failed disk onto a hot spare. A well-designed controller (or set of cooperating controllers) can provide **identical** performance and reliability characteristics for striped mirrors and mirrored stripes.

Mirrored spindles do not help write request performance—they hurt it. It is more than just the fact that every write requests becomes two (or more). It is because, in the safest scenario, individual write completions must wait on all M disks in a mirrored set containing the target data to complete the request. This means the originating write completion is gated on the longer of the two (or more) mirrored writes. This will degrade response times and thereby affect throughput for workloads that have any serialization of disk requests.

On the other hand, read requests receive the benefit of additional data servers (that is, the mirror spindles), and also an intelligent array scheduler can perform optimizations to reduce average read response times by load-balancing and by selecting the best mirror to service any given request.

Replication scorecard (as compared to storing the same amount of data using the same number of disks but in a JBOD configuration):

- For the striping aspect of RAID-0+1, see the "Striping Scorecard" earlier.
- Performance:
  For write requests, both parts of a mirror must be updated, reducing the

available write throughput by >50% for 2-way mirrors, >66% for 3-way mirrors, and so on. On the other hand, read performance improves as the number of mirrors in each set increases.

- Reliability:
  For an M-way mirrored set, all M disks must be lost before data is lost. In the common case of a simple 2-way mirror, MTBF is now increased because it is the longer of two semi-independent disk lifetimes. In most cases, the mirror can be restored quickly (for example, by replacing the disk by hand or using a hot spare), further reducing the window of vulnerability. For non-trivial mirrored striped arrays, most multi-disk failures are survivable.
- Availability:
  If connectivity is lost to one disk, or even M–1 disks of every M-mirrored set in a striped array, work can still proceed, albeit at a different level of throughput; failure of a mirror spindle decreases read performance and increases write performance for that set. When disconnected disks become reachable again, they can be synchronized back to the corresponding active disks, thereby restoring the original array characteristics.
- Capacity/Cost:
  Half the capacity of JBOD for a 2-way mirror, a third for a 3-way mirror, and so on.

Redundancy through Rotated Parity (RAID 5)

This technique provides a certain level of protection from single-disk failures in a striped array. It is provided as an alternative to mirroring with its doubling (or tripling) of storage subsystem cost. In an array of N striped disks, a single disk is added to provide a total capacity of N+1 disks. Parity information is now striped along with the original data on all N+1 disks, providing sufficient information to reconstruct all of the data on any given disk in the array, should it fail or become disconnected. Of course, if two disks fail, all of the data in the array is essentially lost.

Read requests are not greatly affected by the parity data blocks, as they are stripe-unit-sized. That is, a read crossing a stripe unit boundary that leads into a parity area merely results in skipping to the next stripe unit on the next disk. In fact, the additional disk provides a potential increase in array read throughput and can reduce read response times through load-balancing across all N+1 disks.

Write requests, on the other hand, typically suffer a severe performance penalty as compared to a non-redundant striped array. Although large write requests (spanning one or more complete stripes) can write out data and parity blocks in a manner comparable to a striped array, requests smaller than a full stripe or those that do not start and end on stripe boundaries must go through additional steps to update the corresponding parity block (or blocks). This procedure is called a Read-Modify-Write (RMW) operation:

1. Read the data about to be overwritten (the “old” data) and the parity protecting it (the “old” parity).
2. Compute the new parity: $P_{new} = P_{old} \oplus D_{old} \oplus D_{new}$.
3. Write the new data and new parity.

Thus an update to a single stripe unit becomes four requests involving two disks. Furthermore, because the request pairs to each disk consist of a read followed by a write to the same LBN, a full rotation is required between the two—increasing the average rotational delay for the writes as compared to random requests.

It actually takes five or six requests or a power-protected cache and sophisticated recovery mechanisms to insure that a power failure will not result in corrupt data, but most array controllers do not go this far, so this paper only covers the standard 4-request RMW operation.

RAID-5 scorecard (as compared to storing the same amount of data using the same number of disks but in a JBOD configuration):

- For the striping aspect in regards to read requests, see the “Striping Scorecard” earlier.
- Performance:
  For write requests, there can be a *75% or more performance degradation*, depending on the exact array and workload characteristics (request size, temporal and spatial locality, burstiness, stripe unit size, and stripe unit alignment). On the other hand, read performance might improve slightly because there are now N+1 data servers, or it might decrease because the data now has interspersed parity blocks that interrupt streams (somewhat) and extend seek times.

  In degraded mode, when a disk has been lost, write performance actually improves slightly, because in some cases either the parity or the data is unavailable and thus does not have to be read and overwritten. Read performance, on the other hand, is severely degraded, because any requests involving data from the failed disk require the data to be reconstructed by reading an equivalent amount of data from all remaining disks and using the parity to recreate the original data. So an array of N disks with a single failure has at least 1/Nth of its read requests multiplied into N–1 read requests.

  The reconstruction of a new disk full of data requires that every sector in the array be read. This represents a huge drain on the available throughput of the array until the reconstruction is complete. Letting the reconstruction proceed at full speed is the most intrusive choice, but it restores reliability the fastest. Moderating the reconstruction workload to allow reasonable performance for applications has the associated danger of widening the window of vulnerability to additional disk failure and thus total data loss.

  If performance is an important criterion for a storage subsystem, then RAID-5 is rarely the correct solution. But there are exceptions. For example, a workload made up almost entirely of read requests might be reasonably serviced using RAID-5 if the array is not too large or if reverting to a backup (for example, losing today’s updates) is not viewed as a business-breaker. Static web content might be served from a small RAID-5 array, for example, if cost is a major issue and losing a server for any period of time is to be avoided.

  For example, RAID-5 might be the right choice for a write-heavy workload if the following is true:

  - The workload consists of large requests in comparison to the array’s stripe unit size (that is, they constitute several complete stripes of data on average), or the controller has a write-back cache, allowing writes to be delayed and coalesced into full stripes, and
  - The workload is largely sequential (in terms of LBNs), and
  - The array controller can detect and optimize for full stripe writes, and
  - The number of spindles is reasonable, as described in “Rules of Thumb” later in this paper, and
  - Cost is a serious issue.

  In such a case, RAID-5 could be a reasonable choice for storage architecture.

- Reliability:
  If a single disk fails, its data can be reconstructed from the remaining disks' data and parity blocks. But the array is very vulnerable in this state, because it cannot survive another failure. Thus it is critical that another disk be activated (by replacement or hot spare) and the data reconstructed in a timely fashion. The reconstruction performance/reliability trade-off is discussed earlier.

  It is an unfortunate fact that disk failures are not independent, because they can be caused by environmental factors such as heat and vibration or by power issues such as repeated failures, brown-outs, or poorly conditioned power. Disks in an array are subjected to the same environment and same power conditions.

  Perhaps the most worrisome RAID-5 scenario involves "in-flight" write activity at the point of disk failure. It is difficult to determine when data actually reaches the disk media; for example, caches along the way might signal completion in advance of actual media writes. Anything that could possibly leave disk data in a corrupt state might result in additional corruptions due to the relationship created between unrelated disk blocks through shared parity, which could be used in later reconstruction of related blocks. Checking or regenerating all parity after a failure can help prevent data corruption from spreading, but there is a performance cost for reading every sector off every disk in the array.
- Availability:
  A single-disk failure does not prevent access to any of the data, but multi-disk failures are fatal. However, it might be the choice of the system administrator to explicitly stop access to the array during the reconstruction phase so as to complete it as quickly as possible.
- Capacity/Cost:
  An additional disk is added to an array, so the cost is 1/N where N is the number of disks in the original non-redundant array.

Stripe Unit Size Considerations

Not all hardware or software array controllers allow the administrator to choose a stripe unit size; some might provide only a limited set of options. For example, the Windows built-in volume manager for dynamic disks (Dmio.sys) has a fixed stripe unit size of 64 KB. This is unfortunate, because the choice of stripe unit size strongly affects the performance of a disk array. For example, a stripe unit size equal to one 512-byte sector would be inappropriate for a workload consisting of numerous concurrent random 8K requests. Specific examples are given in this section.

For workloads consisting entirely of small random requests, stripe unit size becomes somewhat irrelevant, because load-balancing is already provided by the random workload characteristic. Note that metadata updates must be considered as well. For example, a stripe unit size must be chosen that spreads out updates to the on-disk file system structures if file metadata is being updated frequently.

For workloads with highly variable request sizes and little sequentiality or locality of reference, the choice of stripe unit size should be strongly influenced by access concurrency. The stripe unit size should increase with the expected typical or maximum number of outstanding requests, thereby decreasing the average number of disks accessed per request. [Chen90a]

For example, if the workload is fairly serialized (that is, only a few requests outstanding at any given time), then a smaller stripe unit size will allow multiple disks to be used together to reduce response times for medium and larger requests. But if there are more requests outstanding on average than spindles, a larger stripe unit size will allow each spindle to service a complete request on its own, to maximize concurrency and minimize positioning overhead.

For workloads consisting primarily of aligned, fixed-size requests, the stripe unit size can be chosen such that the number of disk accesses per request is a constant. In particular, using any multiple of the fixed request size allows every request to be serviced by a single disk. This implies that the alignment of the request multiples matches the alignment of the stripe units.

If alignment cannot be guaranteed, then stripe unit size should be increased to reduce the number of times a request crosses a stripe unit boundary. This is because mechanical latencies and processing overheads far outweigh data transfer times for small requests. For more information about stripe unit alignment, see "Diskpar Sample Program" later in this paper.

For workloads with some degree of sequentiality or locality of reference, load balancing must also be considered. When requests are large, load balancing does not significantly affect the choice of stripe unit size. For light workloads, larger stripe units exploit the positioning time benefits of sequentiality and locality of reference. A more complex trade-off exists for general-purpose, commercial workloads, which are often characterized by small requests, bursts of high activity, and biased data-reference patterns. With such workloads, the ability of coarse-grained striping to load-balance an array improves as the stripe unit size decreases. Larger stripe units tend to suffer from the formation of short-term hot spots. Ken Bates suggests that the stripe unit should be approximately 10 times the average request size. [Bates91]

For a workload of low concurrency on a RAID-5 array, the ideal case might be to have a stripe size equal to the file system allocation size. This would remove the penalty of the Read-Modify-Write behavior for writes, because allocation-sized writes will be Full-Stripe writes, and thus parity is calculated anew every time. For this scheme to work, the file system must be aligned on the stripe boundary, not just the stripe unit boundary.

Track size used to be a consideration, but modern hard disks use multi-zone recording. Track capacity changes depending on the radius of the track; outermost cylinders might contain twice as many sectors as innermost cylinders. In addition, the dramatic increase in bits per square inch and resultant decrease in the percent of usable sectors means that more spare sectors are sprinkled throughout the disk, disrupting the sequential layout. Obtaining the exact layout of on-disk data has become difficult if not impossible.

Contributors to the stripe unit size calculation include the following:

- Concurrency:
  The more outstanding requests, the larger the stripe unit size. For example, lots of outstanding requests often implies smaller random requests, which means better efficiency by letting each request be serviced by a single disk (that is, no split requests). Fewer outstanding requests often means larger requests, which are better served by all disks simultaneously, especially if sequentiality is present.

  High concurrency often means that the goal of the storage subsystem should lean towards throughput rather than response time. If the concurrency also includes locality of reference—for example, "hot" files—then the stripe unit size should be chosen as to be small enough to split such hot regions across multiple spindles.

- Sequentiality:
  Smaller stripe unit sizes become more tolerable when the workload has significant sequentiality. This is because the overhead of positioning multiple spindles' heads is amortized over multiple requests to each spindle, aided by automatic prefetching. But if the sequential requests occur in small bursts or occur in conjunction with other request streams, a larger stripe unit size might be more appropriate, so that each burst can take advantage of the prefetching heuristics of a single spindle rather than be spread across multiple spindles, prompting wasted prefetching.
- Burstiness:
  This is really a measure of the dynamic concurrency of the workload. A workload might have N requests outstanding on average, but the reality might be bursts of 10*N requests interspersed with long idle periods. An administrator should optimize a storage subsystem design based on peak workload characteristics as well as averages.
- Read:Write ratio and Read/Write caching:
  For a read workload without read caching, prefetching is not done automatically and sequentiality gains no benefits from staying on a single spindle. For a write workload without write caching, there is no write combining and sequentiality gains no benefit from staying on a single spindle. In fact, it would be better if each sequential write goes to a different spindle to reduce rotational latency—that is, set the stripe unit size to equal the typical write request size. But storage subsystem caching for a workload with locality represents perhaps the most important performance consideration. [Ruemmler94]
- Request size:
  Size has been referred to multiple times in the previous contributor descriptions because it is so crucial. If the workload consists of 64K requests and the array has an 8K stripe unit size, every request involves up to 8 spindles, depending on array size. If, on the other hand, the requests are 8K and the stripe unit size is 64K, then requests (if aligned within the stripe units) will not be split across spindles, but every eighth sequential request will hop spindles. Whether either of these is acceptable or desirable depends on the other factors above. In general, workloads with large requests require the most consideration when it comes to stripe unit sizes, because choosing a small stripe unit size causes each large request to split into multiple disk accesses and thus decrease performance, unless concurrency is very low.
- Alignment:
  If requests are aligned to some value—for example, the allocation unit of a file system—but the alignment does not fall naturally into the stripe units of the array, then even small requests can be split across stripe unit boundaries. This can have a major performance impact, especially if typical requests are not very small in comparison to the stripe unit size. Although this is not the norm, it can happen with some combinations of hardware and software. If requests are aligned, then the stripe unit size should be an integer multiple of the request alignment stride. For information about how to shim the file system (partition) up to a stripe unit or stripe boundary, see “Tools for Storage Performance Analysis” later in this paper.

In an ideal world, the array controller would monitor the characteristics of the workload and the performance of the array, and then either make suggestions to the administrator or automatically change to a better stripe unit size.

Table 1 summarizes the scorecards provided earlier in this section.

Table 1. Disk Array Data Layout and Redundancy Schemes: Pros and Cons

Configuration	Performance	Reliability	Availability	Cost and Capacity
JBOD	Pros: ▪ Concurrent sequential streams to separate disks Cons: ▪ Susceptible to load imbalance	Pros: ▪ Data isolation; single loss affects one disk	Pros: ▪ Single loss does not prevent access to other disks	Pros: ▪ Minimum cost
Striping	Pros: ▪ Balanced load ▪ Potential for better response times, throughput, and concurrency Cons: ▪ Difficult stripe unit size choice	Cons: ▪ Single loss affects entire array	Cons: ▪ Single loss prevents access to entire array	Pros: ▪ Minimum cost
Replication	Pros: ▪ Two data sources for every read request (up to 100% performance boost) Cons: ▪ Writes must update mirrors	Pros: ▪ Single loss and often multiple losses survivable	Pros: ▪ Single loss and often multiple losses do not prevent access	Pros: ▪ Twice the cost
Rotated Parity	Cons: ▪ Up to 75% write performance hit due to RMW ▪ Up to 50% read performance hit in failure mode ▪ All sectors must be read for reconstruction; major slowdown	Pros: ▪ Single loss survivable; “in-flight” write requests may still corrupt Cons: ▪ Multiple losses affect entire array ▪ After loss, array is vulnerable until reconstructed	Pros: ▪ Single loss does not prevent access Cons: ▪ Multiple losses prevent access to entire array ▪ To speed reconstruction, access may be slowed or stopped	Pros: ▪ One additional disk required

Projecting Throughput Requirements

After a system’s aggregate workload has been defined and the potential array configurations have been decided upon, throughput requirements can be projected. Each component of the workload contributes its portion of the throughput, which can be reflected up the I/O path. Given the speeds of the various interconnects, the sustainable throughput can be compared against the needs of the workload.

For example, one component of the workload might be 4K random reads to a RAID-0+1 array of 10 spindles. Given an average and peak interarrival rate or queue length, throughputs (and response times) can be estimated. In most cases, data from existing systems can be gathered to provide the estimates. Array data layout and redundancy schemes must be taken into account, as well as workload characteristics such as locality (sequentiality and cache hit rates).

Windows-related Characteristics and Considerations

A number of considerations for storage subsystem planning and for storage subsystem performance analysis are specific to Windows, although they might have counterparts in other operating systems. This paper addresses just a few of these considerations:

- File system creation and file access
- Storage stack drivers and driver configuration
- Storage commands and request flags

Note: The way Windows handles some storage requests has changed from release to release. This paper was written after the release of Windows Server 2003 and uses that release as the basic model for discussion. Most related functionality in Windows Server 2003 is being released in service packs for Windows 2000 and Windows XP.

File Systems

File System Allocation Size

When creating a new file system, there are trade-offs involved with selecting the appropriate file system allocation size (also known as "cluster size" in NTFS). Considering only the performance issues and assuming a mixture of file sizes, smaller cluster sizes are more efficient in terms of capacity and, in many cases, layout. But if large files are allocated a small piece at a time, fragmentation can cause serious performance degradation. For information about the relationship between file system allocation size and stripe unit size, see "Stripe Unit Size" earlier in this paper.

File Extension

If a large file is allocated a bit at a time, this causes several undesirable performance effects:

- The file will typically be more fragmented.
- The user requests will be synchronous, even if they were requested as asynchronous.
- Additional metadata updates are necessary.
- Multiple searches for free space and space allocation will be performed.

Instead, the file can either be created initially in its entirety (by way of SetFilePointer/SetEndOfFile) or at least extended in large segments, thereby avoiding the majority of the performance degradation.

File Directories and File Names

Although massive directories can be created, eventually performance will degrade due to the size of the data structure used to store the directory metadata. There are a number of ways to alleviate this performance problem:

- Alter the directory structure so that directories are of a more reasonable size in terms of the number of files. For information, see "Rules of Thumb" later in this paper.
- Do not store multiple names for files—for example, eliminate the 8.3 filenames in NTFS. This can be a major performance factor for large directories.

  For NTFS, the **fsutil** utility can be used to query and set the **disable8dot3** parameter, or the registry can be modified directly at HKLM\SYSTEM\CurrentControlSet\Control\FileSystem\NtfsDisable8dot3Name Creation.
- Keep filenames unique, especially at the beginning of the name. For example, do not name files as File0001.txt, File0002.txt, File0003.txt, File0004.txt, and so on.

Miscellaneous suggestions

- Do defragmentation analysis regularly.
- Minimize Last Access Time updates to reduce metadata updates.

  For NTFS, the **fsutil** utility can be used to query and set the **disablelastaccess** parameter, which allows NTFS to delay updating the on-disk version of the value for up to an hour. Using the **disablelastaccess** parameter can affect programs that rely on this feature, such as Backup and Remote Storage.
- Pre-allocate directories and files to improve spatial locality of metadata and file data.

Storage Stack Drivers

Volume Manager (Ftdisk,Dmio)

Ftdisk.sys is the volume manager for basic disks, and Dmio.sys is the volume manager for dynamic disks. In either case, configuring a volume as RAID-0, RAID-1, or RAID-5 will add to the CPU load. RAID-5 is especially CPU intensive, as the new parity must be generated for every Read-Modify-Write or Full-Stripe write. In particular, the dynamic volume manager has not been highly tuned for RAID-5 performance.

SCSIport and Storport

A new option, Storport, has become available for OEM miniport drivers that are used to dealing with the SCSIport interface. Storport provides additional functionality with improved performance. The interface is largely a superset of the SCSIport interface, so porting over existing miniports is not difficult. But the real advantage comes when a miniport uses the new capabilities of Storport—fully duplexed I/O traffic and isolation of I/O setup code from synchronization structures.

NumberOfRequests

Both SCSIport and Storport miniport drivers can use a registry parameter to designate how much concurrency is allowed on a device by device basis. The default is 16, which is much too small for a storage subsystem of any decent size unless quite a number of physical disks are being presented to the operating system by the controller. The maximum value is 255, but this is probably excessive in most configurations. Increasing the value beyond the needed amount results in wasted memory (that is, increased memory footprint).

For information about the optimal number of outstanding requests in a system, see “Rules of Thumb” later in this paper.

Bypassing Process I/O Counts

By default, Windows continuously captures per-process statistics for I/O operations. They can be displayed under the Processes tab in Task Manager or in the Process group in Perfmon. In a multiprocessor environment, this data is shared by the CPUs on which a given process runs. When a process that generates considerable disk and network I/O—such as a database service—runs on several CPUs, updating the shared statistics can slow the system.

The performance reduction can be removed if the system is configured to bypass the per-process I/O counters. To do so:

1. Add the **CountOperations** entry to the registry as a REG_DWORD in HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\I/O System

2. Set the entry value to **0**.

Note: When so configured, Task Manager and Perfmon no longer provide per-process or system-wide I/O statistics.

Flush-Cache, Write-Through, and Write Cache Enable

Windows has become more accurate in terms of correctly supporting flush-cache, write-through, and write-cache-enable features. This can result in what is perceived as a performance regression when upgrading to a new version of the operating system. Provisions have been made in Windows to alleviate such a regression, as discussed in the following sections.

FlushFileBuffers() and IRP_MJ_FLUSH_BUFFERS

Calling FlushFileBuffers() or issuing a FLUSH_BUFFERS IRP through some other technique can result in a complete flushing of all hardware caches between the host and the disk media. The intent of the call might be only to flush dirty data for a given file, but once the IRP is below the file system, there is no way for caches to determine what dirty blocks are associated with that particular file. So the only safe thing to do is to flush all dirty blocks from all caches along the path. This has obvious performance impacts.

Changes have been made to Windows to alleviate some of the performance degradation, such as by collapsing multiple flushes and issuing the minimum number necessary to get the job done. If performance is deemed crucial, or if power-protection is available for protecting dirty data throughout a power failure or system crash, then the new Power-Protected mode is available. For information, see “Power-Protected Mode” later in this paper.

FILE_FLAG_WRITE_THROUGH and ForceUnitAccess

When a write request is tagged with the write-through flag, it is supposed to reach stable storage before the request’s success is reported. Where exactly that stable storage exists might not be important—for example, it could be the disk media or it could be a battery-backed cache along the path to the hard disk.

Write-through requests tend to interrupt streams of (sequential) requests, jerking the hard disk actuator between different regions of the disk and then allowing the stream to pick back up where it left off. A stream of sequential write-through requests is a worst-case scenario; every request will have to wait almost a full rotation of latency before it can write to the media. The semantics of write-through requests means they cannot be combined or else completion responses would be unnecessarily delayed.

The most important issue in regards to write-through requests is that the IDE/ATA protocol supported by current devices does not support this functionality. For SCSI, the ForceUnitAccess (FUA) flag provides the necessary capability. Other protocols have similar capabilities, and FUA has been proposed as an extension to the ATA standard. If performance is deemed crucial, or if power-protection is available for protecting dirty data throughout a power failure or system crash, the new Power-Protected mode is available to the administrator, as described later in this section.

Enabling Write Caching on Disks

The Disk>Properties>Policies dialog allows an administrator to enable write caching, if the operating system determines that such a cache exists on the device.

To be completely cautious, some applications or administrators might disable write caching in an attempt to force all writes to reach the disk media as quickly as possible. Potential problems with this approach include at least the following:

- There is no guarantee that the write caching will actually be disabled. In many cases a controller cache is only configurable using a special utility provided by the manufacturer.
- There are numerous advantages of having a write cache; without a write cache, every request turns into a write-through, with all the disadvantages listed earlier.
- If the cache actually obeys the disable command but it is battery-backed, then the administrator or application has done nothing but hurt performance without increasing reliability.

If write caching is disabled, or if the operating system is not aware of any such cache, then all flush-cache commands and write-through flags are removed before any request is sent out over the wire.

Power-Protected Mode

Windows has been augmented to allow the removal of flush-cache commands and write-through flags on a disk-by-disk basis. The Disk>Properties>Policies dialog has a checkbox below the one that enables write caching. It is clearly marked that this checkbox is intended for storage subsystems with power protection for dirty data (for example, battery-backed caches). When this option is checked, all flush-cache commands and write-through flags are removed before any request is sent out over the wire.

Hardware, Firmware, and Miniport Driver Characteristics and Considerations

This section addresses issues related various peripheral bus protocols, the hierarchical nature of PCI, multiple active cards sharing a peripheral bus, and black box effects.

Torn Writes from a Disk Array Perspective

When power fails on a modern hard disk, the actuator immediately swings the heads over to a safe area. This means that sectors might be half-written, resulting in data loss. Or a subset of sectors for a request might be written and the remaining sectors not written—for example, a torn page. Similarly, controllers do not all guarantee that write requests will be fully completed in the event of a power failure.

Such torn writes require sophisticated recovery algorithms, even in the case of redundant arrays. If a mirrored write completes on only one disk before a power failure, how will the controller know which is the “old” data and which is the “new” data? Or if one of the mirrored writes is torn itself, will the controller handle this scenario correctly?

RAID-5 has an even worse problem, in that corruption to one stripe unit might actually propagate to other stripe units in the same stripe. If an array controller does not check the parity blocks for all stripes that had writes in progress, a torn parity write could result in other stripe units in the same stripe being reconstructed incorrectly if a disk dies at some later date. By putting checksums on pages, database systems and even Microsoft Exchange can detect torn pages, mark them as damaged, and use archived backup copies and change logs to reconstruct the correct pages.

SCSI vs. IDE

Most importantly, an administrator must look at the specifications for the devices themselves, not just make generalizations based on protocol, or even cost. Large on-board caches can result in a big performance win. Seek time comparisons should include (if possible) average, single-cylinder, and full-stroke seeks. An administrator should attempt to determine what is meant by the term "average seek," because it can be interpreted several ways. Rotational speed is more important than seek speeds for many workloads, with modern disks ranging from 5400 RPM to 15000 RPM. The number of heads means less than it used to, although more heads might mean that there are fewer cylinders and therefore shorter seeks on average.

Historically there were some inherent disadvantages to IDE/ATA: Concurrency was severely limited, write-through was not supported, and raw bus speeds were suboptimal. Many modern IDE controllers "pretend" to be SCSI to enable request concurrency and function at a level closer to that of their SCSI and Fibre Channel (FC) counterparts. Although command queuing is an important requirement for high performance, the outstanding ATA issue is the lack of ForceUnitAccess functionality. If the storage subsystem architecture is such that dirty data is power-protected, this might not be a major issue.

Cost and throughput are still major differentiators for IDE and SCSI/FC hard disks. High-speed SCSI disks service 150 to 250 random requests per second, with IDE disks servicing roughly half as many in the same period of time. But SCSI disks can cost up to five times that of IDE disks. This brings capacity into consideration as well if one decides to save money by buying, for example, four IDE disks rather than two SCSI disks. Not only does the buyer hope to obtain SCSI-level performance, but might realize a capacity increase. Or if the workload is not entirely random, having four actuators rather than two might provide additional performance benefits.

PCI and PCI-X Buses

Given the throughput of current and future network cards (1–20 Gb/s) and disk controllers (quad-ported high-speed SCSI or Fibre Channel), there is serious competition for resources along the I/O path. The PCI hierarchy is one such bone of contention. Administrators should have detailed knowledge of the PCI hierarchy within each system and the expected load from each PCI card. Juggling cards can provide significant improvement for an unbalanced I/O-bound system. Cases have been observed where throughput on a 66-MHz 64-bit PCI bus goes from 100 MB/s to 250 MB/s.

Tools such as Perfmon can be used to obtain throughput information. Some of the PCI hierarchy can be inferred from device properties data, but the machine specifications are the best source. Sometimes the hierarchy is actually printed on the system chassis near the PCI card slots.

Competition for PCI Resources

PCI and now PCI-X have not kept up with the throughput now possible from some network cards and disk controllers. As a result, issues can arise when two active cards share a bus.

Multiple cards represent an opportunity to spread the workload across multiple PCI buses (as described later in this section). On the other hand, a single 4-way card could mean fewer PCI arbitration "collisions," assuming two or more cards ended up on the same bus and therefore the possibility of longer transactions and better PCI utilization, assuming decent intelligence in the card. On the other hand, if there are other active PCI agents, it means that a fair arbiter will grant a single card less of the available bandwidth than four cards.

Multiple cards means a single stream to one set of disks will not trash the cache of the other three sets; a single 4-way card means the total cache resources must be used dynamically by all four sets.

Multiple cards most likely result in fairness of bandwidth from channel to memory, as the PCI arbiter and memory bridges are usually fair; a single 4-way card might or might not be fair among its channels or might exhibit strange interleaving.

The way interrupt vectors/pins are assigned and the way the Interrupt Service Routines (ISRs) work might give an advantage to one scheme over another. That is, one does not want a scenario where all four cards (or all four channels) need be checked when an interrupt is detected—the driver should know exactly which card/channel did it immediately.

Multiple cards mean less opportunity for interrupt-combining/reduction, but if the 4-way cannot handle interrupt combining, it might be limited in terms of the number of interrupts per second.

Command processing at multiple cards can obviously occur in parallel (and also concurrent cache usage); on a 4-way card, the implementation controls how much concurrency is possible. On the other hand, four times the amount of silicon/dollars are being thrown into the 4-way controller logic— it might end up being more efficient.

Multiple cards might mean software-managed arrays, if redundancy or striping is advantageous. A single 4-way card might already be capable of handling multi-bus arrays, or it could be replaced with such a card without disturbing the rest of the PCI hierarchy.

This section might seem vague, with too many variables to consider. The best way to find out what works best is to use a PCI analyzer to observe what is actually happening on the bus.

Black Box Effects

To an operating system, a storage subsystem is at one or more levels a black box. “Disks” might not really be single spindles. There might be a variety of caching (power-protected or not), redundancy schemes, and layout heuristics behind each “disk.” Caches might be disabled, or they might falsely report their state. Write-through flags and flush-cache commands might be dropped, for example, so that the device exhibits better performance characteristics.

Without opening the black box, administrators might make decisions that they would never make if they had full knowledge. It is rarely possible to obtain all of the useful information, but an administrator interested in storage performance should gather as much data as possible to make an informed decision.

Tools for Storage Performance Analysis

This section describes the following tools and utilities:

Perfmon—Windows Performance Monitor
Exctrlst Utility
Kernrate and Kernrate Viewer
Rescan Utility
Chkdsk Utility
Diskpar Utility (for hard disks with 32-bit Windows)

Perfmon—Windows Performance Monitor

Perfmon is provided with Windows. A number of its performance counters are useful for storage performance analysis: Physical Disk and Logical Disk counters, plus counters such as % DPC Time and Interrupts/sec from the Processor performance object.

Logical Disk and Physical Disk Counters

The same counters are valuable in each of these counter objects. Logical disk data is tracked by the volume manager (or managers), and physical disk data is tracked by the partition manager.

Note: If the Windows standard stacked drivers scheme is circumvented for some controller, so-called “monolithic” drivers might take on the role of partition manager or volume manager. In these cases, it is up to the monolithic driver writer to supply the same counters through the Windows Management Instrumentation (WMI) interface.

- **% Disk Read Time, % Disk Time, % Disk Write Time, % Idle Time**
  These counters are of little value when there are multiple spindles behind “disks.” Imagine a subsystem of 100 disks, presented to the operating system as 5 disks (each backed by a 20-disk RAID-0+1 array). Now imagine that the administrator spans the five physical disks with one logical disk (volume X:). One can assume that any serious system needing that many spindles will have at least one request outstanding to X: at any given time. This will make the “disk” appear to be 100% busy and 0% idle, when in fact the 100-disk array might be up to 99% idle.
- **Average Disk Bytes / { Read | Write | Transfer }**
  Useful in gathering average, minimum, and maximum request sizes. If the sample time is long enough, a request size histogram can be generated. If possible, workloads should be observed separately; multi-modal distributions cannot be differentiated using Perfmon if the requests are consistently interspersed.
- **Average Disk Queue Length, Average Disk { Read | Write } Queue Length**
  Useful in gathering concurrency data, including burstiness and peak loads. For a discussion on what constitutes excessive queuing, see “Rules of Thumb” later in this paper. These values represent the number of requests in-flight below the driver taking the statistics. This means the requests are not necessarily queued but could actually be in service or completed and on the way back up the path. Possible in-flight locations include the following:
  - Sitting in a SCSIport or Storport queue

  - Sitting in a queue at an OEM driver
  - Sitting in a disk controller queue
  - Sitting in an array controller queue
  - Sitting in a hard disk queue (that is, on-board a real spindle)
  - Actively receiving service from a hard disk
  - Completed, but not yet back up the stack to the point where the statistics are gathered
- **Average Disk sec / {Read | Write | Transfer}**

  Useful in gathering disk request response time data, and possibly extrapolating service time data. These are probably the simplest indicators of storage subsystem bottlenecks. For a discussion on what constitutes excessive response times, see "Rules of Thumb" later in this paper. If possible, workloads should be observed separately; multi-modal distributions cannot be differentiated using Perfmon if the requests are consistently interspersed.
- **Current Disk Queue Length**

  An instantaneous measurement of the number of requests in flight and thus subject to extreme variance. As such, not of much use except to check for the existence of numerous short bursts of activity, which is lost when averaged over the sample period.
- **Disk Bytes / second, Disk {Read | Write } Bytes / second**

  Useful in gathering throughput data. If the sample time is long enough, a histogram of the array's response to specific loads (queues, request sizes, and so on.) can be analyzed. If possible, workloads should be observed separately.
- **Disk {Reads | Writes | Transfers } / second**

  Useful in gathering throughput data. If the sample time is long enough, a histogram of the array's response to specific loads (queues, request sizes, and so on.) can be analyzed. If possible, workloads should be observed separately.
- **Split IO / second**

  Only useful if the value is not in the noise. If it becomes significant, in terms of Split I/Os per second per spindle, further investigation could be warranted to determine the size of the original requests being split and the workload generating them.

Processor Counters

- **% DPC Time, % Interrupt Time, % Privileged Time**

  If Interrupt Time and Deferred Procedure Call Time are a large portion of Privileged Time, the kernel is spending a significant amount of time processing I/Os. In some cases, it works best to keep interrupts and DPCs affinitized to a small number of CPUs on a multiprocessor system, in order to improve cache locality. In other cases, it works best to distribute the interrupts and DPCs among many CPUs, so as to keep the interrupt and DPC activity from becoming a bottleneck.
- **DPCs Queued / second**

  Another measurement of how DPCs are consuming CPU time and kernel resources.
- **Interrupts / second**

  Another measurement of how Interrupts are consuming CPU time and kernel resources. Modern disk controllers often combine or coalesce interrupts so that a single interrupt results in the processing of multiple I/O completions. Of course, it is a trade-off between delaying interrupts (and thus completions) and economizing CPU processing time.

Exctrlst—Extensible Performance Counter List

This utility can be used to enable or disable performance counters in a system-wide manner. For example, PerfDisk{.DLL} is the Library File / Service / Counter ID for Logical Disk and Physical Disk counters.

This utility is available in the Windows Server 2000 and Windows Server 2003 Resource Kits.

Kernrate and Kernrate Viewer

Profiling tools such as Kernrate and KrView can be used to determine whether a specific driver is taking up excessive CPU time and also to identify what code within the driver is responsible.

Kernrate is available in the Windows Server 2003 Resource Kit, and Kernrate Viewer, along with a viewer-compatible version of Kernrate, can be downloaded from the Web at http://www.microsoft.com/whdc/hwdev/platform/performance/default.mspx.

Rescan Utility

This sample program queries SCSI and IDE devices on the system for basic information such as Bus, ID, Type, Manufacturer and Model, and Firmware revision. For devices such as hard disks, it also gives information on whether the device uses command queuing, linked commands, or a synchronous interface. It also provides the width of the interface. Note that this is not an exhaustive list of the output.

Rescan is available as a download from Knowledge Base at: http://support.microsoft.com/default.aspx?scid=kb;en-us;308669

Chkdsk Utility

Chkdsk, which is provide with Windows, scans a file system to check for inconsistencies in the file system metadata (and, if requested, the data sectors). Chkdsk can be configured through various parameters to perform cursory or exhaustive analyses. It has gone through significant performance enhancements since Windows 2000 was first released. For some workloads, an order of magnitude improvement can be observed going from Windows 2000 to Windows XP, and again from Windows XP to Windows 2003 Server.

Chkdsk is faster now, but some basic directory layouts can cause Chkdsk to take more time. First, directories containing huge numbers of files are more difficult to process efficiently than directories of a more reasonable size. The exact dividing point is, of course, dependent on the system configuration. For approximate numbers, see “Rules of Thumb” later in this paper.

Also, the use of multiple file names (for example, 8.3 file names in NTFS) can significantly extend the run time of Chkdsk.

Diskpar Sample Program (for hard disks with 32-bit Windows)

Diskpar finds and modifies the starting sector on a disk to fix mismatches between file system allocation units and stripe units. If the file system is not aligned correctly, requests that are only the size of a single file system allocation unit might still be split between disks. Diskpar has also been used on occasion to adjust the alignment of file system allocation units so that they match storage hardware cache line boundaries.

The Master Boot Record (MBR) on a disk formatted using a 32-bit version of Windows limits the maximum default hidden sectors to 63. For this reason, the default starting sector for disks that report more than 63 sectors per track is the 64th sector. For disks reporting 64 or more sectors per track, this can cause misalignment when the disk participates in a striped layout. For example, with a disk that reports 64 sectors per track, what appear to be track-aligned transfers begin at the last sector of one track and span to the next track. But that is not the real problem, because track information reported by disks usually does not reflect reality. Most disks have different numbers of sectors on different tracks—for example, outer versus inner bands.

The same data is reported by a disk array, even though the concept of a "track" is somewhat meaningless in this context. But misalignment can still exist between the file system allocation units and the internal stripe unit boundaries.

So primarily Diskpar is used to nudge a file system starting position so that it is aligned to meet caching or striping requirements. But without detailed information at the hardware level—for example, using a hardware analyzer—it will most likely be a trial and error experiment. The good news is that there are only seven different starting positions before "wrapping around" if using a 4K file allocation unit size, and 15 if 8K, and so on. The only reason to go beyond this small number of values is if the administrator needs to align the file system to a stripe boundary instead of a stripe unit boundary. For workloads that profit from this layout, see "Stripe Unit Size" earlier in this paper.

Any disk request generator that provides a heavy workload of small concurrent requests can be used to test whether Diskpar adjustments succeed in reducing the number of requests crossing stripe unit boundaries.

The Diskpar sample program is available in the Windows Server 2000 and Windows Server 2003 Resource Kits.

Rules of Thumb

This section includes these topics:

Stripe Unit Size
Mirroring versus RAID-5
Response Times
Queue Lengths
Number of Spindles per Array
Files per Directory

Stripe Unit Size

See "Stripe Unit Size Considerations" earlier in this paper for a full description of all of the factors that go into this decision, including array controller limitations, metadata writes versus data writes, file allocation unit alignment, RAID-5 considerations, concurrency, sequentiality, burstiness, read:write ratio, read and write cache sizes and heuristics, and request size.

Response time rule of thumb with no workload data available:

Several storage researchers have suggested 256 KB as the minimum stripe unit size for a modern Windows system.

Mirroring Versus RAID-5

RAID-5 was a reasonable solution a decade ago, when cost and capacity were serious considerations. With the reduction in price per spindle and the dramatic increase in capacity per spindle, there are few environments where RAID-5 provides the best trade-off between reliability/availability and performance. For a workload with any reasonable fraction of writes, the read-modify-write performance hit is overwhelming compared to the cost of purchasing the additional spindles for mirroring. The susceptibility to multi-disk failures and the chance of corruption spreading beyond the initially affected sectors further reduces the attraction of RAID-5. Finally, the vast majority of RAID-5 controllers do not provide the kind of security that is associated with this scheme by the average system designer or administrator.

Redundancy rule of thumb with no workload data available: RAID 0+1 in the form of striped mirrors.

Response Times

Tools such as Perfmon can be used to obtain data on disk request response times. Write requests hitting a write-back hardware cache often have extremely low (<1ms) response times, because completion is virtually instantaneous and the data is written back to disk media in the background. As the workload begins to saturate the cache, response times will climb until the write cache's only benefit is a better ordering of requests to reduce positioning delays.

For JBOD arrays, reads and writes are roughly equivalent in performance characteristics. Positioning delays for random requests are on the order of 5–15ms for modern hard disks. Positioning delays for sequential requests should be insignificant except in the case of write-through streams, where each positioning delay should approximate the time necessary for a single disk rotation.

Transfer times are usually less significant in comparison to positioning delays, with the exception of sequential requests and large requests (>256K) that are instead dominated by disk media access speeds as the requests become larger or more sequential. Modern hard disks access their media at 15–50 MB per second depending on rotation speed and sectors per track, which varies from region to region on a given disk model. For random I/O, the shared buses used by a hardware-managed disk array might be lightly utilized; but for sequential or large requests, a bus or even the controller itself could become the limiting factor.

For striped arrays, if the stripe unit size is well chosen, each request will be serviced by a single disk—except in the case of low concurrency workloads. So the same general positioning and transfer times still apply. For software-managed arrays, a comparison of Logical Disk and Physical Disk counters, specifically reads per second and writes per second, can give insight into how many requests are crossing stripe unit boundaries.

For mirrored arrays, write completions might need to wait for both disks to complete the request. Depending on how the requests are scheduled, the two requests could complete quite a while apart. However, though writes generally should not take twice the time to complete for mirrored arrays, they will probably be slower than JBOD. Reads, on the other hand, can experience a nice performance boost if the array controller is dynamically load-balancing or scheduling requests to the disk that will incur the least positioning delay.

For RAID-5 arrays (rotated parity), writes turn into four separate requests in the typical Read-Modify-Write scenario. This is roughly the equivalent of two "mirrored" reads plus a full rotation of the disk, assuming the RW pairs proceed in parallel. For software-managed arrays, comparing the writes per second performance counters for Logical Disks and Physical Disks shows the read-modify-write request multiplier.

The performance impact of redundant arrays on read and write requests must be taken into account when planning subsystems or analyzing performance data. For example, Perfmon might show 50 writes per second being processed by volume X:, but in reality this could mean 100 requests per second for a mirrored array, or 200 requests per second for a RAID-5 array, or even more than 200 requests per second if they are being split across stripe units.

Another example would be the response time for RAID-5 writes, which might include two random read requests, a delay while waiting for both read requests to complete and parity to be calculated, and two write requests requiring a full rotation of positioning time. The only thing that can make this look like a quick process is a write-back cache with lots of time for background accesses.

Response time rule of thumb with no workload data available:

For a lightly-loaded system, average write response times should be less than 1 millisecond if a large write cache is available, less than 15 milliseconds on non-RAID-5, and less than 25 milliseconds on RAID-5. Average read response time should be less than 15 milliseconds.

For a heavily-loaded system that isn’t saturated, average write response times should be less then 50 milliseconds on non-RAID-5, and less than 75 milliseconds on RAID-5. Average read response time should be less than 50 milliseconds.

Queue Lengths

There are a number of viewpoints regarding what constitutes excessive disk request queuing. The one espoused by this paper can be defined as follows: The boundary between a busy disk subsystem and a saturated one is a persistent average of two requests per spindle. A disk subsystem is “perfectly busy” when every spindle is always servicing a request and every spindle has at least one queued-up request in order to maintain concurrency—that is, to keep the pipeline moving. Disk requests split into multiple requests (due to striping or redundancy maintenance) count as multiple requests in this rule of thumb.

Of course, there are caveats to this rule. Administrators probably do not want all their spindles busy all of the time, even though that is the “perfect” scenario above. But because disk workloads are normally bursty, it is far more likely that this rule will be applied over shorter periods of (peak) time. Requests are typically not uniformly spread among all hard disks at any given time, so the administrator must take into account deviations between queues—especially for bursty workloads. One should also recognize that a longer queue provides more opportunity for disk request schedulers to reduce positioning delays. Array controllers can also take advantage of longer queues to optimize for Full-Stripe RAID-5 writes or mirrored read selection.

There are multiple queues along the I/O path—inside drivers, disk controllers, array controllers, and hard disks themselves. The higher along the path, the harder it is to establish exactly how requests are distributed among spindles; thus the rule becomes fuzzier. At the same time, variance decreases, so the rule has a better chance of working overall.

Because hardware has more capability to queue up requests—either through multiple queuing agents along the path or just agents with more queuing capability—increasing the multiplier threshold might allow more concurrency within the hardware, although there is a potential increase in response time variance. Ideally the additional queuing time is balanced by increased concurrency along with service time optimizations enabled by longer queues.

Queue length rule of thumb with little workload data available:

For a lightly-loaded system, average queue length should be less than 1 per spindle with occasional spikes up to 10. If the workload is write-heavy, the average queue length above a mirrored controller should be less than 0.6 per spindle and less than 0.3 per spindle above a RAID-5 controller.

For a heavily-loaded system that isn't saturated, average queue length should be less than 2.5 per spindle with infrequent spikes up to 20. If the workload is write-heavy, the average queue length above a mirrored controller should be less than 1.5 per spindle and less than 1 above a RAID-5 controller.

Number of Spindles per Array

For striped arrays, the trade-off is between data isolation (small arrays) and better load-balancing (large arrays). For RAID-1 arrays, the trade-off is between better cost/capacity (mirrors—that is, depth of 2) and the ability to withstand multi-disk failures (shadows—that is, depths of 3 or even 4). Read and write performance issues can also play a role in RAID-1 array size. For RAID-5 arrays, the trade-off is between better data isolation and MTBF (small arrays) and better cost/capacity (large arrays).

Given that hard drive failures are not independent, array sizes must be limited when the array is made up of actual spindles (that is, a bottom-tier array). It is extremely difficult to say what that limit should be.

Array size rule of thumb with no hardware reliability data available:

Bottom-tier RAID-5 arrays should not extend beyond a single desk-side storage tower or a single row in a rack-mount configuration. This means approximately 8–14 spindles for modern storage enclosures.

Bottom-tier mirrored arrays should not extend beyond two towers or rack-mount rows, with data being mirrored between towers or rows when possible. These rules of thumb help to avoid or alleviate the decrease in MTBF caused by using multiple buses, power supplies, and so on from separate storage enclosures.

Files per Directory

File creation, access, and deletion times are affected by the number of files in the target directory. The directory data structure containing the filenames must be searched when creating, accessing, or deleting a file or executing Chkdsk or similar utilities. The larger the data structure, the longer most searches will take. A number of experiments have shown that directories containing tens of thousands of files do not experience significant performance issues. Even directories containing a few hundred thousand files might perform well for some operations. But beyond that point, performance will almost always begin to suffer. So it is important to design an application or file system layout that does not require having millions of files in a single directory.

Eliminating multiple filenames (for example, 8.3 filenames in NTFS) has a large beneficial effect on performance for large directories, effectively increasing the number of files that can be stored without experiencing a performance hit. To remove 8.3 filesnames already in existence, backup, clean, and immediately restore the affected directories.

Files per directory rule of thumb with no workload data available:

Don't place more than 250,000 files in a single directory, and if possible reduce that limit to 100,000. Unless absolutely necessary, disable 8.3 filenames.

Summary and Future Work

Analyzing storage subsystem performance is an art, not a science. Performance is balanced in a delicate trade-off against reliability, availability, and cost (capacity). Many choices are driven by workload characteristics such as concurrency, sequentiality (locality), burstiness (interarrival patterns), read:write ratio, request size, and alignment. Others are driven by hardware considerations such as caches, power protection, failure rates, and peripheral bus characteristics.

Each component along the path has limited information available to it for use in optimizing for performance or other criteria. Windows provides a number of tools for gathering performance data and adjusting performance parameters.

Rules of thumb discussed in this paper include caveats for virtually every storage guideline. The best outcome of reading this report is for system administrators to find out everything they can about their workloads and their hardware before making decisions or in preparation for future performance analysis.

Many facets of storage subsystem performance have not been covered in this paper. Future updates to this paper may include the following topics:

- Components of a modern hard disk
- HyperTransport
- Software-managed versus hardware-managed arrays
- SCSI or IDE (direct-attached storage) versus Fibre Channel
- Network-attached storage (NAS) versus storage area networks (SAN)
- Disk request generators
- Detailed tracing and displaying of disk request activity

References

Storport documentation on MSDN
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/storage/hh/storage/portdg_83s7.asp

[Anderson03] D. Anderson, J. Dykes, E. Riedel, "More Than an Interface—SCSI vs. ATA," Seagate Research. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, California, March/April 2003.

[Barclay03] T. Barclay, W. Chong, J. Gray, "A Quick Look at SATA Disk Performance," MSR-TR-2003-70, Microsoft Research, October 2003.

[Bates91] K. Bates, *VAX I/O Subsystems: Optimizing Performance*, Professional Press Books, Horsham, Pennsylvania, 1991.

[Bitton88] D. Bitton, J. Gray, "Disk Shadowing," *Proceedings of the 14th International Conference on Very Large Data Bases*, Long Beach, California, September 1988, pp. 331-338.

[Chen90] P. Chen, G. Gibson, R. Katz, D. Patterson, "An Evaluation of Redundant Arrays of Disks Using an Amdahl 5890," *Proceedings of the 1990 ACM*

SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Boulder, Colorado, May 1990, pp. 74-85.

[Chen90a] P. Chen and D. Patterson, “Maximizing Throughput in a Striped Disk Array,” *Proc. 17th Int’l Symp. Computer Architecture,* IEEE CS Press, Los Alamitos, California, Order No. 2047, 1990, pp. 322-331.

[Chung00] L. Chung, B. Worthington, R. Horst, J. Gray, “Windows 2000 Disk IO Performance,” MSR-TR-2000-55, Microsoft Research, June 2000.

[FAST] Various papers from the *Proceedings from 1st and 2nd USENIX Conferences on File and Storage Technologies*.

[Ganger93] G. Ganger, Y. Patt, “The Process-Flow Model: Examining I/O Performance from the System’s Point of View,” *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Santa Clara, California, May 1993, pp. 86-97.

[Ganger94] G. Ganger, B. Worthington, R. Hou, Y. Patt, “Disk Arrays: High-Performance High-Reliability Storage Subsystems,” *IEEE Computer*, Vol. 27, No. 3, March 1994, pp. 30-36.

[Gray90] J. Gray, B. Horst, M. Walker, “Parity Striping of Disk Arrays: Low-Cost Reliable Storage with Acceptable Throughput,” *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990, pp. 148-161.

[Gray00] J. Gray, P. Shenoy, “Rules of Thumb in Data Engineering,” MSR-TR-99-100, Microsoft Research, December 1999. *Proceedings of the 16th International Conference on Data Engineering*, San Diego, California, March 2000, pp. 3-12.

[Patterson88] D. Patterson, G. Gibson, R. Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID),” *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, May 1988, pp. 109-116.

[Ruemmler94] C. Ruemmler, J. Wilkes, “An Introduction to Disk Drive Modeling,” *IEEE Computer*, Vol. 27, No. 3, March 1994, pp. 17-28.

[Worthington96] B. Worthington, G. Ganger, Y. Patt, J. Wilkes, “On-Line Extraction of SCSI Disk Drive Parameters,” CSE-TR-323-96, University of Michigan, December 1996.