A DCD Filter Driver for Windows NT^{TM} 4

Xubin He and Qing (Ken) Yang
Department of Electrical & computer Engineering
University of Rhode Island
Kingston, RI 02881
e-mail: {hexb, qyang}@ele.uri.edu

Abstract

With the rapid increases in processor speed, disk I/Os will eventually become a system bottleneck. We have recently proposed a new disk I/O architecture called DCD (*Disk Caching Disk*) that can drastically improve disk I/O write performance as shown by simulation experiments [2,4]. To validate whether DCD can live up to its promise in the real world, this paper presents a design and implementation of one DCD configuration as a filter driver under Windows NT, one of the most popular operating systems in today's computing world. Performance measurements have been carried out using synthetic benchmarks consisting of random file writes. Experimental results show that DCD can improve the synchronous write performance by a factor of up to 13.8 for intensive small write requests. In addition, because DCD uses part of hard disk as disk cache, it has excellent reliability both for meta-data and user data. Furthermore, the DCD driver is completely transparent to the upper file system (NTFS) and the lower level physical device. It does not require any modifications to the original OS nor the existing on-disk layout. As a result, it can be inserted into the existing NT driver hierarchy to obtain immediate performance and reliability improvements.

1 Introduction

With the rapid advances in semiconductor technology, the speed disparity between CPU and disks increased dramatically during the past decade. As a result, disk operations often dominate users' waiting time in today's commercial computing environments. As one of the leading operating systems, Windows NT has a number of nice features specifically targeted for enhancing performance of its file system (NTFS) and disk I/Os [5]. For example, NTFS improves performance by using a write-back caching strategy. That is, it writes the modifications to the cache and flushes the cache to disk as a background thread. NTFS provides file system recovery based on a transaction-processing model in which each I/O operation that alters file system data or volume's directory structure is considered as a transaction. Every such transaction is logged as a log record in a log file (Figure 1).

Riedel et al [3] have recently proposed several techniques to further enhance the performance of NTFS such as using large requests, bypassing the file system cache, spreading the data across many disks and controllers, and using deep-asynchronous requests to achieve the best performance [3].
While the NTFS out-of-the-box performance is quite good, the overheads for small write requests can be quite high. Current NTFS' recoverability guarantees only that the meta data and volume structures not be corrupted. Protection of user data in the event of a system crash is not guaranteed

This paper presents a different and new approach for improving disk I/O performance under Windows NT systems. The technology is called DCD (*Disk Caching Disk*) that we have recently invented as a new hierarchical disk architecture that can potentially improve disk write performance by 2 orders of magnitudes. The idea of the DCD is fairly simple. DCD converts small requests into large ones before writing data in to a disk. A similar approach has been successfully used in database systems [7]. Simulation results show that DCD has the potential for drastically improving write performance (for both synchronous and asynchronous writes) with very low extra cost. We have designed and implemented a DCD device driver for Windows NT operating system. Measured performance results show that the DCD driver runs significantly faster than the traditional system while ensuring file system integrity in the events of system crashes. For small and bursty writes, our DCD driver improves synchronous writes by a factor of 13.8 in terms of response time seen by users. DCD driver can also improve the reliability of the system because both meta data and user data are cached in a disk that is much more reliable than RAM. Moreover, the DCD driver is completely transparent to the OS. It does not require any changes to the OS. It does not need accesses to the kernel source code. And it does not change the on-disk data layout. As a result, it can be used as a ``drop-in" replacement of the traditional disk device driver in an existing system to obtain immediate performance and reliability improvements.

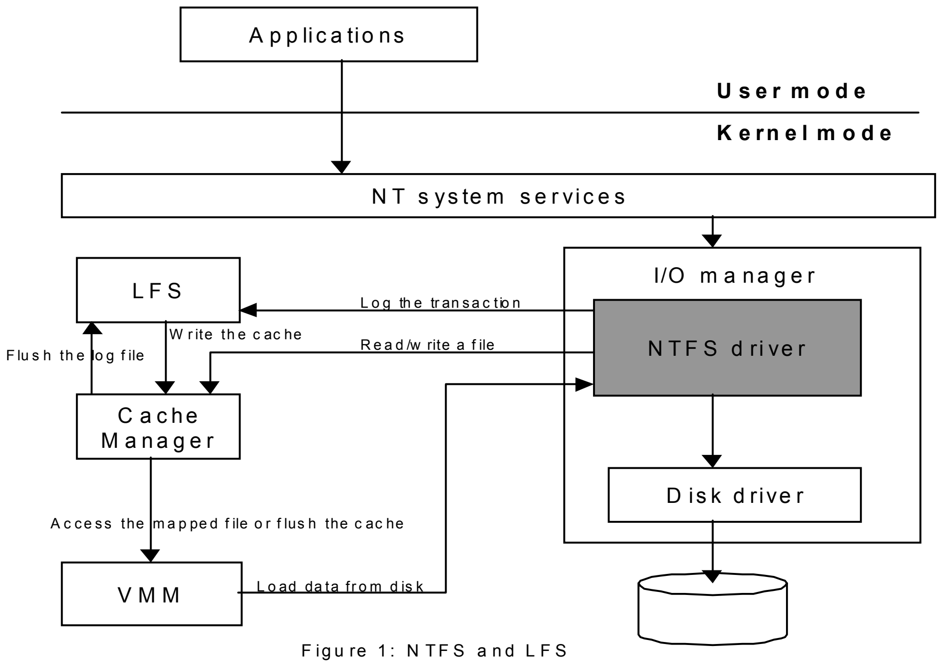


Figure 1: NTFS and LFS

The paper is organized as follows. The next section gives an overview of the DCD architecture and its basic operations. Section 3 presents the detailed descriptions of the design of the DCD filter driver on Windows NT. Experimantal settings and measured performance results are presented in Section 4. We conclude the paper in Section 5.

2. Disk Caching Disks

Figure 2 shows the general structures of DCD. The fundamental idea behind DCD is to use a small log disk, called *cache-disk*, as an extension of a small NVRAM (Non-Volatile RAM) buffer on top of a *data-disk* where a normal file system exists. The NVRAM buffer and the cache-disk together form a two-level cache hierarchy to cache write data. Small write requests are first collected in the small NVRAM buffer. When the NVRAM buffer becomes full, the DCD writes all the data blocks in the RAM buffer, *in one large data transfer,* into the cache-disk. This large write finishes quickly since it requires only one seek instead of tens of seeks. As a result, the NVRAM buffer is very quickly made available again to accept new incoming requests. The two-level cache appears to the host as a large virtual NVRAM cache with a size close to the size of the cache-disk. When the data-disk is idle or less busy, it performs *destaging* operations which transfer data from the cache-disk to the data-disk. The destaging overhead is quite low because most data in the cache-disk are short-lived and are quickly overwritten therefore requiring no destaging at all. Moreover, many systems, such as those used in office/engineering environments, have sufficient long idle periods between bursts of requests. Destaging can be performed in the idle periods therefore will not interfere with normal user perations at all.

Since the cache in DCD is a disk with a capacity much larger than a normal NVRAM cache, it can achieve very high performance with much less cost. It is also non-volatile thus highly reliable.

The cache-disk in DCD can be a separate physical disk drive to achieve high performance (a *physical DCD*) as shown in Figure 2 (a), or one logical disk partition physically residing on one disk drive for cost effectiveness (a *logical DCD*) as shown in Figure 2 (b). In this paper we concentrate on implementing the logical DCD, which does not need a dedicated cache-disk. Rather, a partition of the existing data-disk is used as a cache-disk. In addition, the RAM buffer shown in Figure 2 (b) is implemented using a small portion of the system RAM. Therefore this implementation is complete software without any extra hardware. It should be noted that many systems have two or more disk drivers. In these cases, if we let the logical cache-disk partition on one disk cache data on another disk, it may achieve performance close to that of a physical cache-disk. Because of load imbalance in many systems, the data disk and the disk that contains the cache-disk partition often are not likely to be active at the same time therefore they will not interfere with each other.

One very important fact is that the cache-disk of DCD is only a cache that is completely transparent to the file system. Hu and Yang pointed out that DCD can be implemented at the device or device driver level [2], which has many significant advantages. Since a device driver is well-isolated from the rest of the kernel, there is no need to modify the OS and no need to access the kernel source code. In addition, since the device driver does not access kernel data structures, it is less likely that any kernel changes will require a re-implementation of the driver. Furthermore, since DCD is transparent to the file system and does not require any changes to the on-disk data layout, it can easily replace a current driver without affecting the data already stored on the disk (as long as a free cache-disk partition somewhere in the system can be provided). This is very important to many users with large amount of installed data.

DCD can speed up both synchronous writes and asynchronous writes. Although asynchronous writes normally do not have direct impacts on the system performance, they often affect the system performance indirectly. For example, Windows NT systems flush dirty data in the file system cache to disks periodically for reliability reasons, creating a large burst of overhead writes. This large burst will interfere with the normal read and write activities of users. DCD can avoid this problem by using a large and inexpensive cache-disk to quickly absorb the large write burst, destaging the data to the data-disk only when the system is idle.

Using results of trace-driven simulations, Hu and Yang have demonstrated that DCD has superior performance compared to traditional disk architectures in terms of I/O response times by using trace-driven simulations[2]. However, trace-driven simulations have limited accuracy because there are no feed backs between the traces and the simulators. The main objective of this research is to examine whether DCD will have good performance in the real world. We believe that the best way to evaluate the real world performance is to actually implement the DCD architecture and measure its performance in terms of user response times, using some realistic benchmarks.

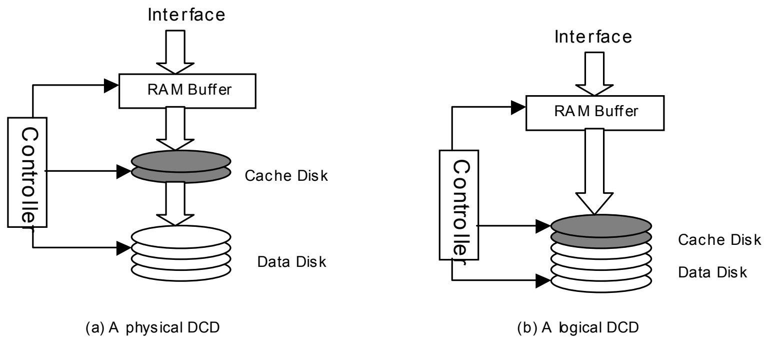


Figure 3: The structure of DCD

3. Design and Implementation

This section describes the data structures and algorithms used to implement our DCD device driver. Please note that while we have adopted the DCD architecture from [2], our implementation algorithms are considerably different from the ones reported in [2]. We choose these new algorithms mostly because they offer better performance and simplified designs. In some cases a new algorithm was chosen because of the different design goals of the DCD device driver and the original DCD. For example, the original DCD uses an NVRAM buffer and it reorders data in the NVRAM buffer to obtain better performance. However, when implementing the DCD driver, we have to use the system DRAM as the buffer. We can not reorder data in the DRAM buffer before the data is written to the disk, otherwise we can not maintain the order of updates to the disk. Keeping the order has some impact on the performance, but it is the price that has to be paid for maintaining the file system integrity. Figure 4 shows the DCD driver in the Windows NT driver hierarchy. Our DCD driver is a filter driver in the driver hierarchy which intercepts the IRPs and provide value-added functionality not supplied with the based Windows NT while keeping its original features.

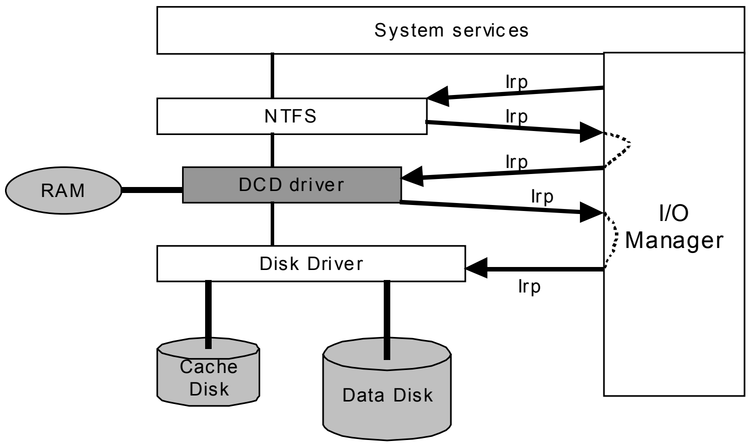


Figure 4: DCD driver on the NT driver Hierarchy

3.1 Driver device object

Device object structures are created by kernel-mode drivers to represent logical, virtual or physical devices. Any physical devices are represented bin memory by device objects. For example, \Device\Floppy0 and \Device\Harddisk0\Partition0 are the name of the internal Windows NT device objects representing the floppy disk drive and the first hard disk drive respectively. Without a device object, a kernel-mode driver will not receive any I/O requests, since there must be a target device for every I/O request dispatched by the I/O manager. A driver may have multiple device objects associated with it. Our DCD driver will create a device object associated with it by issuing a call to IoCreateDevice( ) where an additional amount of nonpaged memory is allocated and associated with the newly created device object. This memory is called the device object extension which structure is as the following:

```
typedef struct DEVICEEXTENSION{
PDEVICE_OBJECT DeviceObject;    //A back pointer to itself
PDEVICE_OBJECT TargetDeviceObject;  //Pointer to the data disk device object
PDEVICE_OBJECT CacheDeviceObject;  //Pointer to the cache disk device object
PUCHAR   RAMImage;                //RAM buffer for DCD
……
PUCHAR   RAMHeader;               //In memory header of RAM buffer
ULONG    RAMSize;                 //Size of RAM buffer
ULONG    CacheDiskSize;           //Size of Cache Disk
}
```

Where RAMImage represents the RAM buffer in DCD hierarchy, and it is an amount of continuous memory allocated from the system nonpaged pool by ExAllocatePool(NonPagedPool, DCD_RAM_SIZE).

3.2 RAM image

RAM image is a reserved, nonpaged memory allocated when the DCD driver is loaded during the system boot. It consists of two main parts, RAM Headers and Segment buffers. RAM headers are the in-memory copies of the segment header of the cache disk which will be discussed later. Each segment buffer consists of a buffer header and several slots. Each segment has its own unique Segment Buffer ID. Our driver makes the segment buffers appear infinite by reusing it circularly (while guaranteeing that it does not overwrite information it needs). A slot is the basic caching unit (Figure 5). In our implementation a segment buffer size is 64KB and a slot can be used to store 1KB data.

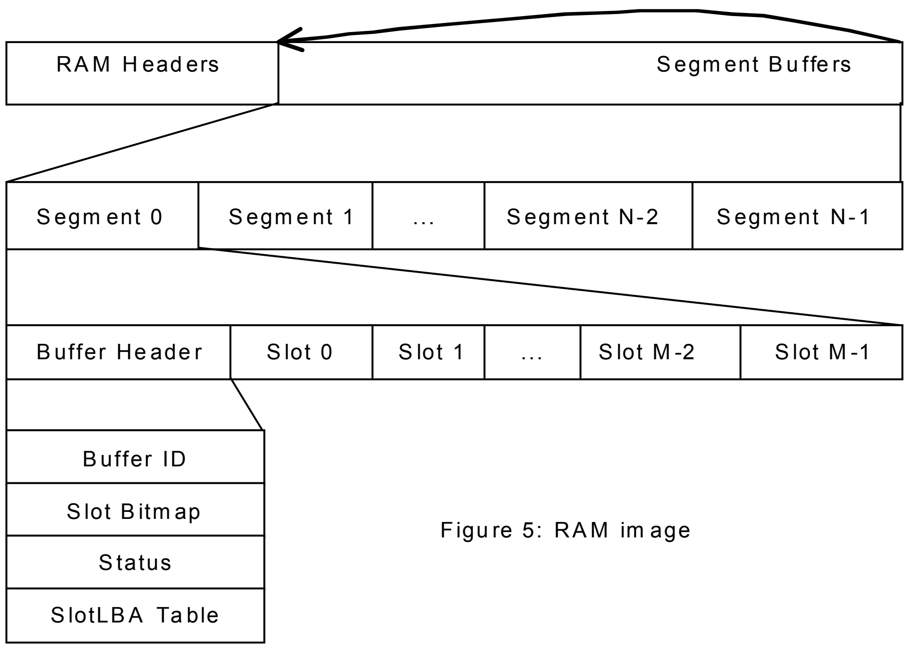


Figure 5: RAM image

The buffer header describes the contents and the status of the corresponding segment buffer. It contains:
1) A *Buffer ID* which is used to identify a segment buffer.
2) A *Slot bitmap* which is used to describe the status (free or used) of each slot within the segment buffer.
3) A *Status* field to describe the status of the segment buffer. A segment may have one of the 3 statuses: *free* which means no data on this segment; *dirty* which means that the segment contains data which is not written to the cache disk yet; *valid* which means that the segment contains data which has already been written to the cache disk.
4) A *SlotLBA table*. Each slot in the segment has a corresponding entry in the table. Each entry is an integer which describes the target Logical Block Address (LBA) of the data in the slot.

Initially all Segment Buffers are free. When a write request arrives, the driver picks a free Segment Buffer to become the *Current Segment Buffer*. Meanwhile the driver also obtains a free disk segment from the Free-Segment-List to become the *current Disk Segment*. The driver then ``pairs'' the RAM segment buffer and the disk segment together by writing the Segment Buffer ID to the corresponding field in the RAM header. From this point until the current Segment Buffer is written into the current Disk Segment on the cache-disk, incoming write data is written into the slots of the Segment Buffer.

3.3 Cache-disk Organization

The entire cache disk is divided into a number of fixed-size segments (clusters). The DCD driver always writes an entire segment to the cache disk. The organization of the cache disk is similar to that of the RAM image. The cache disk consists of segments, and each segment also has a header (*segment header*) and several slots. The segment size and slot size are 64KB and 1KB respectively. The segment header contains the following information:

1) A *Segment ID* which identifies the segment.
2) A *Buffer ID* which associates the segment with the RAM segment buffer.
3) A *Slot bitmap* which is used to describe the status (free or used) of each slot within the segment.
4) A *Time Stamp* which records the time when the segment is written into the cache disk. During a crash recovery period, the tim stamps help the DCD driver to search for segments.
5) A *SlotLBA* table which function is the same as the table in the Buffer header.

The segment headers in the cache disk (On-Disk-Headers) are only for crash recovery purpose and are never accessed during normal operations. Once a segment is written to the cache disk, the contents of its segment header are fixed. However, some of its slots will be overwritten by subsequent requests therefore the contents of the header should be changed accordingly. Because of this, each segment header has an in-memory copy (In-Memory-Header) which is stored in the RAM Headers area of the RAM image. Once a segment is written into the cache-disk, only the In-Memory-Header is updated by overwriting. The On-Disk-Header is left untouched. If the system crashes, the In-Memory-Headers can be rebuilt by scanning the cache-disk and ``replaying" the On-Disk-Headers ordered by their time stamps. Because the On-Disk-Headers have fixed locations on the disk, this scanning process can be done very quickly.

3.4 Destages

The RAM buffer and the cache disk are used as cache memory in DCD, and eventually data on them will have to be destaged to their original locations on the data disk. There are two kinds of destage operations, that is, destaging data from RAM buffer to the cache disk and destaging data from the cache disk to the data disk.

3.4.1 Destage data from RAM buffer to Cache Disk

A destage thread will be activated when one of the following situations occurs:

- The request size is larger than the free RAM buffer size when a new WRITE request comes in; or
- When the DCD driver detects an idle period; or

For the first situation, the destage thread will continue to move the data from RAM buffer to cache disk until the new request can be satisfied; as the second situation, the destage thread will continue until the RAM buffer becomes empty, or until a new request comes in.

3.4.2 Destage data from cache disk to data disk

The destage thread will start when the DCD driver detects an idle period and no dirty data on the RAM buffer, i.e., no data are needed to be destaged from the RAM buffer to the cache disk. Once the destage thread starts, it will continue until the cache-disk becomes empty, or until it is interrupted by a new request. The original DCD uses a “last-write-first-destage” algorithm [2]. While this algorithm has some advantage, because of the temporary locality, the newly written data has a higher chance of being overwritten shortly, this algorithm may results in unnecessary destaging work since it always destages the newly written data first. A destaging algorithm called “least-cost” algorithm is adopted in [6] where the smaller block is destaged first because segments with more valid data have more chances to be invalidated by overwriting.
We use the simple “first-write-first-destage” algorithm.

3.5 Crash Recovery

In case of system failure, NTFS cannot guarantee protection of user data but it guarantees that the meta-data and volume structures won’t be corrupted. So what we need to do is to destage the data left on the cache disk when the system crashes to the data disk. Since cached data are save reliably on the cache disk, user data will not be lost.

4. Performance Results

In this section, we report the measurement results of our filter driver on Windows NT.

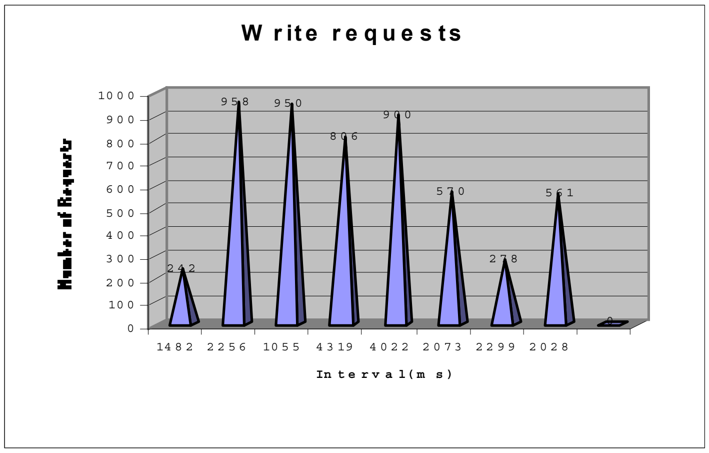


Figure 6. Disk request paterns.

4.1 Workload Characteristics

In order to evaluate the performance of the new filter driver that we have implemented, performance measurements have been carried out. Workload of the measurement plays an important role in the evaluation. With lack of comprehensive benchmarks that represent typical workload for commercial applications, we have studied several real world I/O traces such as Cello, Snake, hplajw, and EMC-tel obtained from HP lab and EMC. One important characteristic of all these traces is that I/O requests are bursty. When there is a request, it is usually accompanied by a cluster of requests in a short time frame. It is common to find more than 5 requests waiting in a queue and the maximum queue length goes up to 100 even 1000. In addition, there is usually a relatively long period of idle time between two consecutive request bursts.

Based on the above observation, we have generated a sequence of synthetic workload. The system issues a sequence of burst requests to the disk I/O system. The number of requests in each burst and the interval time between two subsequent request bursts are both modeled using random numbers. The random numbers are generated using C language functions: srand( ) and rand(). Figure 6 shows the typical request sequences over time. Since we are interested in small write performance, the request sizes vary from 1 KB to 4 KB.

Processor	Dell Dimension XPS 400 Pentium II 400MHZ						
Cache	L1: 32KB (16K data+16K instruction) L2: 512KB						
Memory	64MB SDRAM						
Disk	Model	Interface	Capacity	RPM	Seek	Transfer(MB/s)	Cache
	DTTA 371010	ATA-4	10GB	7200	9.5ms	8-13	512KB
OS	Microsoft Windows NT Server 4.0 SP3, NT File System						

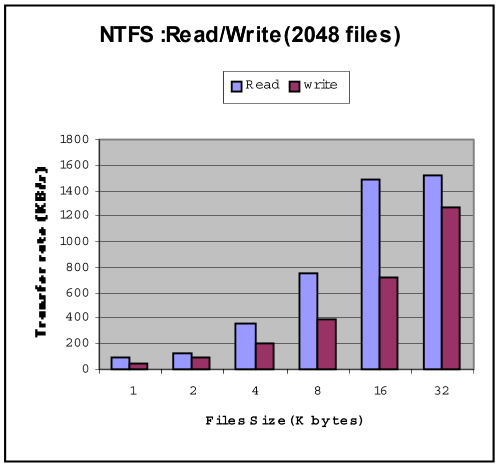
Table 1: Basic Hardware and Software Configuration

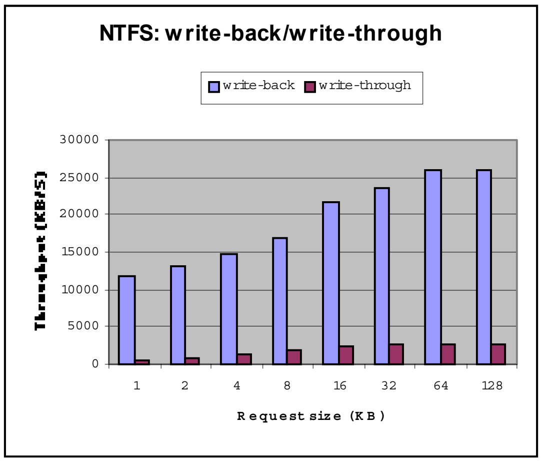
4.2 Experimental Setup

Our filter driver has been developed and tested on Dell Dimension XPS 400 PC with Pentium II 400MHZ processor running Windows NT server. One megabytes of system RAM was allocated specifically for the DCD configuration as the RAM buffer in the three level hierarchy. Twenty megabytes disk partition is used as cache disk and the remaining disk space is used as data disk.The response time is measured by the C language function GetTickCount( ). Table 1 lists the hardware and software configurations of the computer system used for our measurements.

4.3 Overall performance of Current Disk I/Os

First of all, let us take a close look at the out-of-the-box performance of synchronous reads and writes using the NTFS defaults. A simple application that uses the NT file system to read and write files was run and timed. Figure 3 shows the throughput of the file system as a function of file size in KB. The first bar graph (Figure 3a) shows the throughput for the case where the application writes and reads a specific number (2048) of small files with fixed file size. As can be seen from this figure, the system throughput is very low for small file sizes because of the intensive meta data accesses. In Figure 3b, throughputs are shown for both write-back (data are written into RAM cache first and flushed to disk later) and write-through (every write goes through the disk) modes. Write-through is forced by turning on the FILE_FLAG_WRITE_THROUGH when opening a file with CreateFile( ) function. It can be seen from this figure that while write-back performance is fairly good, write-through performs very poorly indicating the presence of disk bottleneck. As we indicated previously, we also observed that the throughput reduces as the request size gets smaller.





(a) (b)

Figure 6. NTFS write/read performance

4.4 Performance Results and Comparison

Our first experiment is to measure the I/O throughput of the system with the new DCD filter driver and compare with that of system without the DCD driver. Figure 7 shows the measured results. For small request size, we notice significant performance improvement as we predicted. As the file size increases, the performance improvement reduces and becomes almost identical when the request size exceeds 64KB since the DCD bypass all requests that are greater than 64 KB. These measurement results agree with our expectation. As we discussed preciously, the performance gain of the DCD mainly results from combining small disk accesses into large ones to minimize seek and rotation latencies. It can be noted from this figure that the throughput improvement can be significant for small requests.

Although Figure 7 show certain performance improvements, it is far less than our previous simulation results and our measurement results under the Unix OS [6]. One of the reasons why we observed limited performance gain is that our filter driver has to go through the I/O Manager before actual disk operations, as shown in Figure 4. As a result, the driver may have to go through a number of layers and queues that are not necessary. In order to bypass the I/O Manager below the DCD filter driver, we created a 20 MB virtual disk in the system RAM. The DCD driver has a direct access to the virtual disk after all the driver functions. The access times of the virtual disk is set to be exactly the same as the disk including seek time, rotation latency, and data transfers.

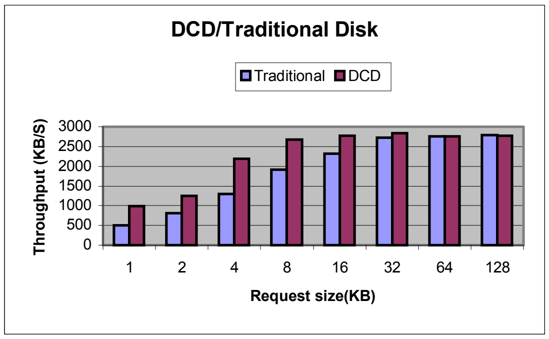
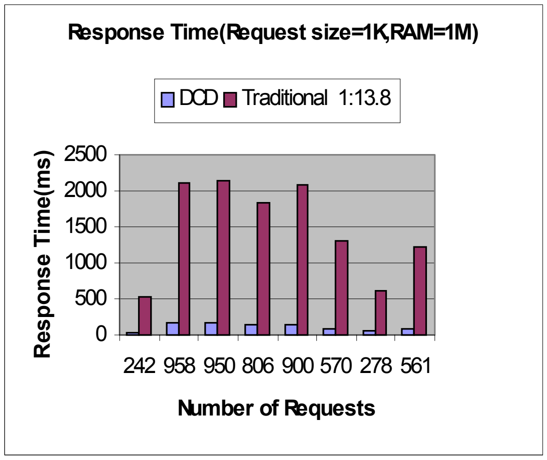
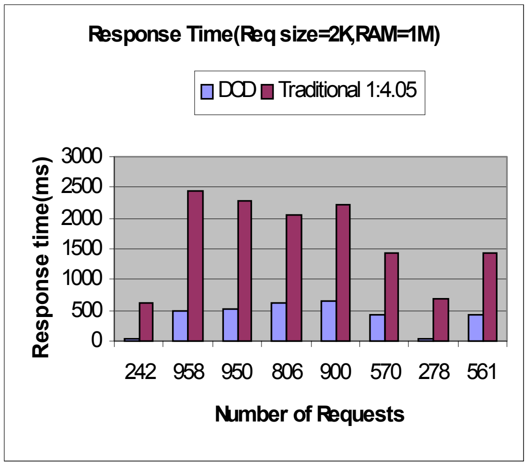


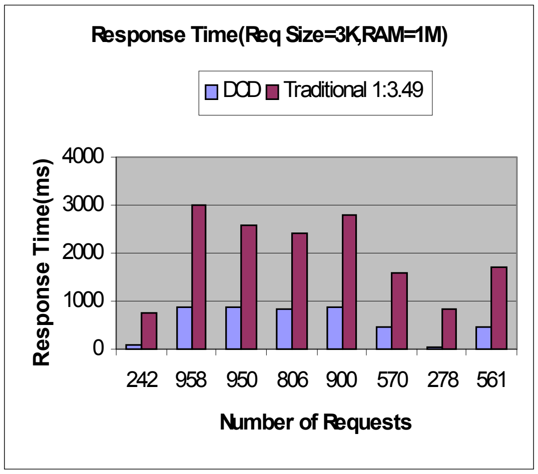
Figure 7. Throughput comparison between the DCD driver performance and current disk performance.



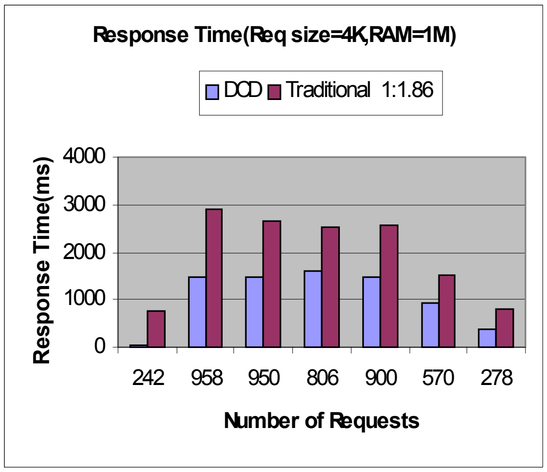
(a)



(b)



(c)



(d)

Figure 8. Performance comparison in terms of I/O response time between the system with the DCD filter driver and the system without the DCD driver.

Figure 8 shows the performance comparison between the system with the DCD filter driver and the system with no DCD driver assuming synchronous writes. In this figure, response times as seen by a user are plotted as a function of number of disk requests in each request burst. Four bar graphs show the performances for four different request sizes of 1KB, 2KB, 3KB and 4KB, respectively. We observed significant performance improvement with the DCD filter driver as shown in these figures. For request size of 1KB, The speed up goes to as high as a factor of 13.8. For other request sizes, the speed up are 4.05, 3.49, 1.86 for request size being 2 KB, 3KB, and 4KB, respectively.

5. Summary and Conclusions

We have designed and implemented a Disk Caching Disk (DCD) filter driver for the Windows NT operating system. Experimental results show that DCD can improve the synchronous write performance by a factor up to 13.8 for the intensive small write requests and improve the reliability of asynchronous write. Moreover, the DCD driver is completely transparent to the upper file system (NTFS) and the lower physical device driver. It does not require any modifications to the original OS nor the existing on-disk layout. It therefore can be inserted into the existing NT driver hierarchy to obtain immediate performance and reliability improvement.

The DCD driver is implemented as a filter driver which is located under the NTFS in the windows NT driver hierarchy. The DCD driver intercepts the requests from the upper level and calls a low-level disk device driver through Windows NT I/O manager to perform actual disk I/O operations. While this approach improves the portability, the potential performance gain of DCD is limited by the I/O manager and the low-level disk-specific disk drivers. As a future research we plan to implement DCD as a SCSI driver with a RAM buffer built in the disk controller.

References

[1] Rajeev Nagar, Windows NT File System Internals: A Developer's Guide, ISBN 1-56592-249-2, O'Reilly & Assocates, 1997.

[2] Y. Hu and Q. Yang, DCD-disk caching disk: A New Approach for Boosting I/O Performance, 23rd Annual International Symposium on Computer Architecture, Philadelphia PA May, 1996, pp.169-178

[3] Erik Riedel, Catharine van Ingen, Jim Gray, A Performance Study of Sequential I/O on Windows NT^{TM} 4, Proceedings of the 2nd USENIX Windows NT Symposium , Seattle, WA, August 1998

[4] Y. Hu and Q. Yang, A New Hierarchical Disk Architecture, IEEE Micro, Vol. 18, No. 6, November/December 1998.

[5] David A. Solomon, Inside Windows NT Second Edition, Microsoft Press, 1998

[6] Tycho Nightingale, Y. Hu and Q. Yang, The Design and Implementation of a DCD Device Driver for Unix, 1999 USENIX Technical Conference, June 6-11, 1999, Monterey, CA, June, 1999

[7] Klaus Elhardt, Rudolf Bayer, A Database Cache for High Performance and Fast Restart in Database Systems. TODS 9(4): 503-525(1984)