Improving Database Performance With AIX Concurrent I/O

A case study with Oracle9*i* Database on AIX 5L version 5.2

Authors: **Sujatha Kashyap**
Bret Olszewski
Richard Hendrickson
{skashyap, breto, richhend}@us.ibm.com

1 Introduction

A new file system feature called "Concurrent I/O" (CIO) was introduced in the Enhanced Journaling File System (JFS2) in AIX 5L™ version 5.2.0.10, also known as maintenance level 01 (announced May 27, 2003). This new feature improves performance for many environments, particularly commercial relational databases. In many cases, the database performance achieved using Concurrent I/O with JFS2 is comparable to that obtained by using raw logical volumes. This paper details the implementation and operational characteristics of Concurrent I/O, and presents the results of our performance evaluation of Concurrent I/O with Oracle9*i* Database.

The file system has long been the heart of UNIX® storage management. Commands and interfaces for manipulating and managing data stored on files are commonly used throughout the UNIX world by users of all skill levels. Managing persistent data via such universally understood mechanics is key to application portability. File systems thus provide a very useful and desirable abstraction for data storage.

As is often the case with any method of abstraction, however, the use of file systems results in some tradeoffs between performance and ease of use. The fastest means of transferring data between an application and permanent storage media such as disks, is to directly access more primitive interfaces such as raw logical volumes. The use of files for data storage involves overheads due to serialization, buffering and data copying, which impact I/O performance. Using raw logical volumes for I/O eliminates the overheads of serialization and buffering, but also requires a higher level of skill and training on the part of the user since data management becomes more application-specific. Also, while file system commands do not require system administrator privileges, commands for manipulating raw logical volumes do. However, due to its superior performance, database applications have traditionally preferred to use raw logical volumes for data storage, rather than using file systems.

With the Concurrent I/O feature now available in JFS2, database performance on file systems rivals the performance achievable with raw logical volumes.

2 Using File Systems for Database Applications

For database applications, the superior performance of raw logical volumes compared to file systems arises from certain features of the file system:

- The file buffer cache
- The per-file write lock, or inode lock
- The sync daemon

These file system features help ensure data integrity, improve fault tolerance, and in fact improve application performance in many cases. However, these features often pose performance bottlenecks for database applications. This section explains the role of these features in a file system, how they impact database performance, and the options provided by JFS2 to help reduce their performance impact.

2.1 File Buffer Cache

At the most basic level, a file is simply a collection of bits stored on persistent media. When a process wants to access data from a file, the operating system brings the data into main memory, where the process can examine it, alter it, and then request that the data be saved to disk. The operating system could read and write data directly to and from the disk for each request, but the response time and throughput would be poor due to slow disk access times. The operating system therefore attempts to minimize the frequency of disk accesses by buffering data in main memory, within a structure called the file buffer cache. On a file read request, the file system first attempts to read the requested data from the buffer cache. If the data is not already present in the buffer cache, it is read from disk and cached in the buffer cache. Figures 1 and 2 show the sequence of actions that take place when a read request is issued under this caching policy.

Similarly, writes to a file are cached so that future reads can be satisfied without necessitating a disk access, and to reduce the frequency of disk writes. The use of a file buffer cache can be extremely effective when the cache hit rate is high. It also enables the use of sequential read-ahead and write-behind policies to reduce the frequency of physical disk I/O's.

Another benefit is in making file writes asynchronous, since the application can continue execution without waiting for the disk write to complete. Figure 3 shows the sequence of actions for a write request under cached I/O.

While the file buffer cache improves I/O performance, it also consumes a significant portion of system memory. AIX's Enhanced JFS, also known as JFS2, allows the system administrator to control the maximum amount of memory that can be used by the file system for caching. JFS2 uses a certain percentage of real memory for its file buffer cache, specified by the **maxclient%** parameter. The value of **maxclient%** can be tuned via the **vmo** command. By default it is set to 80, which implies that JFS2 can use up to 80% of real memory for its file buffer cache. The range of acceptable values for **maxclient%** is from 1 to 100. For example, the following command will reduce the maximum amount of memory that can be used for the file buffer cache to 50% of real memory: **vmo –o maxclient%=50**.

In contrast, raw logical volumes do not use a system-level cache to cache application data, so there is neither duplication nor double-copying of data.

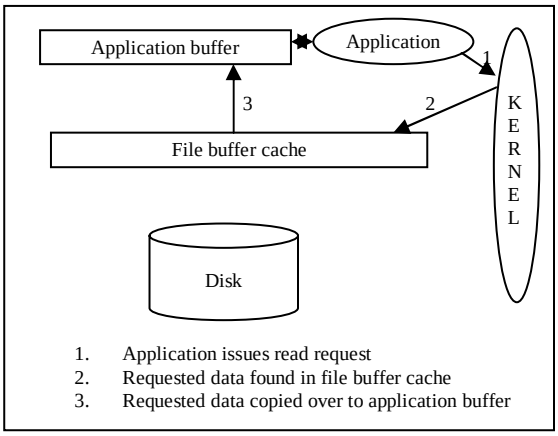

Figure 1: Reads under cached I/O – buffer cache hit

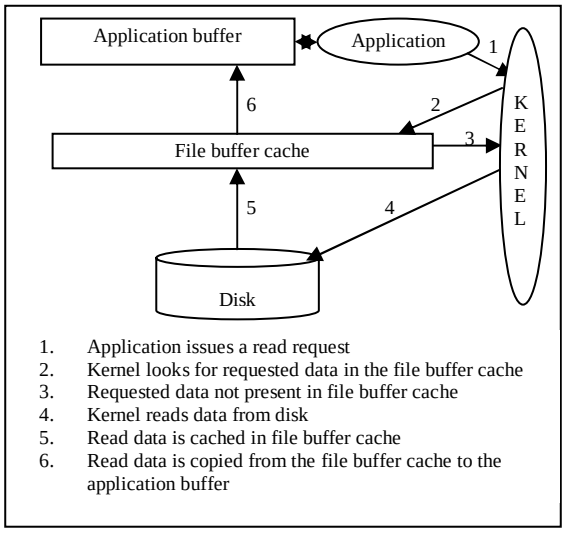

Figure 2: Reads under cached I/O - buffer cache miss

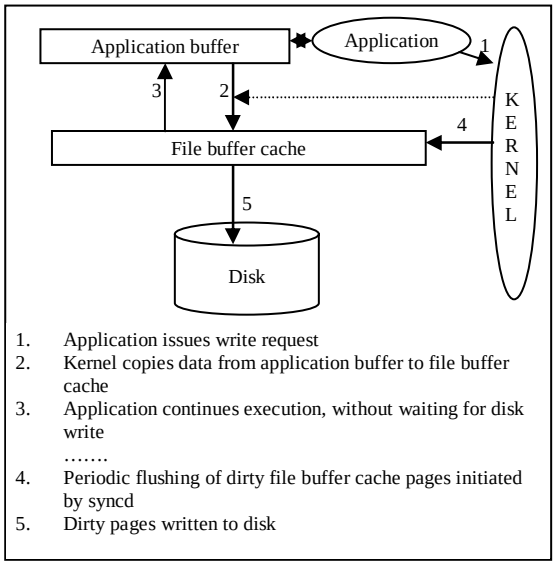

Figure 3: Writes under cached I/O

2.1.1 Direct I/O

Certain classes of applications derive no benefit from the file buffer cache. Some technical workloads, for instance, never reuse data due to the sequential nature of their data accesses, resulting in poor buffer cache hit rates. Databases normally manage data caching at the application level, so they do not need the file system to implement this service for them. The use of a file buffer cache results in undesirable overheads in such cases, since data is first moved from the disk to the file buffer cache and from there to the application buffer. This "double-copying" of data results in additional CPU consumption. Also, the duplication of application data within the file buffer cache increases the amount of memory used for the same data, making less memory available for the application, and resulting in additional system overheads due to memory management.

For applications that wish to bypass the buffering of memory within the file system cache,

Direct I/O is provided as an option in JFS2. When Direct I/O is used for a file, data is transferred directly from the disk to the application buffer, without the use of the file buffer cache. Figures 4 and 5 depict the sequence of actions that occur for reads and writes under Direct I/O.

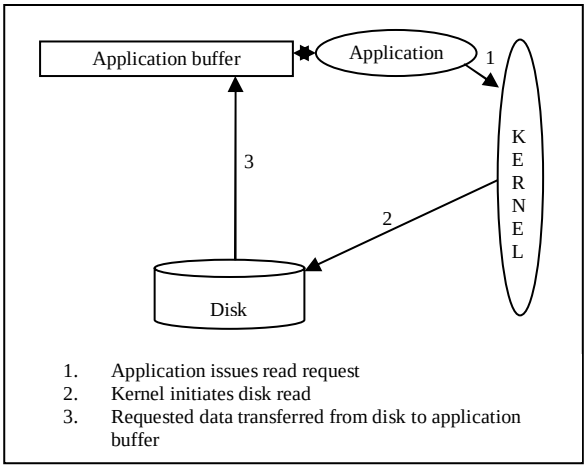

Figure 4: Reads under Direct I/O

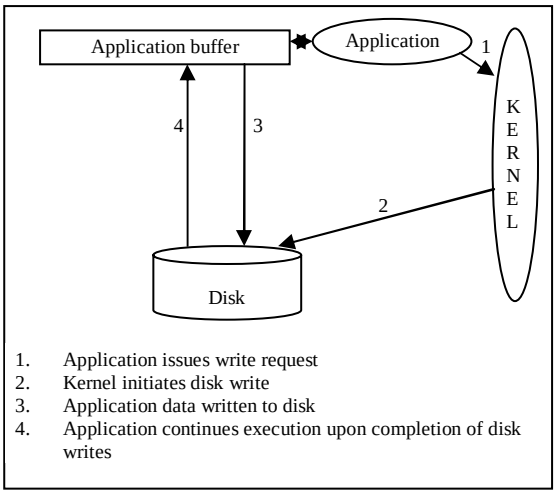

Figure 5: Writes under Direct I/O

2.1.1.1 Direct I/O Usage

Direct I/O can be used for a file either by mounting the corresponding file system with the **mount –o dio** option, or by opening the file with the **O_DIRECT** flag specified in the **open()** system call. When a file system is mounted with the **–o dio** option, all files in the file system use Direct I/O by default. Direct I/O can be restricted to a subset of the files in a file system by placing the files that require Direct I/O in a separate subdirectory and using **namefs** to mount this subdirectory over the file system. For example, if a file system *somefs* contains some files that prefer to use Direct I/O and others that do not, we can create a subdirectory, *subsomefs*, in which we place all the files that require Direct I/O. We can mount *somefs* without specifying **–o dio**, and then mount *subsomefs* as a **namefs** file system with the **–o dio** option using the command: **mount –v namefs –o dio /somefs/subsomefs /somefs**.

The use of Direct I/O requires that certain alignment and length restrictions be met by the application's I/O requests. Table 1 lists these requirements for JFS2. Failure to meet these requirements causes reads and writes to be done using normal cached I/O, but after the data is transferred to the application buffer, the cached copy is discarded. File system read-ahead does not occur for files that use Direct I/O.

To avoid consistency issues, if there are multiple processes open a file and one or more processes did not specify O_DIRECT while others did, the file stays in the normal cached I/O mode. Similarly, if the file is mapped in memory through the **shmat()** or **mmap()** system calls, it stays in normal cached mode. Once the last conflicting, non-direct access is eliminated (by using the **close()**, **munmap()**, or **shmdt()** system calls), the file is moved into Direct I/O mode. The change from caching mode to Direct I/O mode can be expensive because all modified pages in memory will have to be flushed to disk at that point.

Table 1: JFS2 restrictions for Direct I/O

File system format	Buffer alignment	Buffer length increment
JFS2 before AIX 5.2 ML01	4K bytes	4K bytes
JFS2 as of AIX 5.2 ML01	*agblksize* specified at file system creation	*agblksize* specified at file system creation

2.1.1.2 Performance Considerations Under Direct I/O

Direct I/O benefits applications by reducing CPU consumption and eliminating the overhead of copying data twice – first between the disk and the file buffer cache, and then from the file

buffer cache to the application's buffer. However, several factors could impact application performance when Direct I/O is used.

Every Direct I/O read causes a synchronous read from disk, unlike the normal cached I/O policy where the read may be satisfied from the file buffer cache (refer Figures 2 and 5). This can result in poor performance if the data was likely to be in memory under the normal caching policy.

Direct I/O also bypasses JFS2 read-ahead. File system read-ahead can provide a significant performance boost for sequentially accessed files. When read-ahead is employed, the operating system tries to anticipate future need for pages of a sequential file by observing the pattern in which an application accesses the file. When the application accesses two successive pages of the file, the operating system assumes that the program will continue to access the file sequentially, and schedules additional sequential reads of the file. These reads are overlapped with application processing, and will make the data available to the application sooner than if the operating system had waited for the program to access the next page before initiating the I/O. The number of pages to be read ahead is determined by two parameters:

- **j2_minPageReadAhead**
  Number of pages read ahead when the operating system first detects sequential access. If the program continues to access the file sequentially, the next read-ahead is twice **j2_minPageReadAhead**, the next for 4 times **j2_minPageReadAhead**, and so on until the number of pages reaches **j2_maxPageReadAhead**. Default value is 2.
- **j2_maxPageReadAhead**
  Maximum number of pages the operating system will read ahead in a sequential file. Default value is 8.

These parameters are tunable, and can be set via the **ioo** command.

Table 2 compares the performance of Direct I/O versus cached I/O for three different read scenarios. The file block size used in these experiments was 4K bytes, and the default values of **j2_minPageReadAhead**=2, and **j2_maxPageReadAhead**=8 were used.

The first row in Table 2 corresponds to the case where the application reads a 1MB file sequentially, byte by byte. When Direct I/O is used in this case, the alignment restrictions are violated. Consequently, normal cached I/O is used to read a 4K page into the file buffer cache, the requested byte is copied from the file buffer cache to the application buffer, and the 4K page is discarded from the file buffer cache. This results in a 4K page being read for every byte requested by the application, while also incurring the costs of double-copying of data. Cached I/O in this case enjoys two advantages: the 4K page that is brought into the file buffer cache when a single byte is read can be re-used to return 4K bytes of data to the application upon subsequent read requests. Additionally, read-ahead would occur with cached I/O, further reducing the latency of future read requests.

The second row in Table 2 corresponds to the case where a 1GB file is read sequentially in 4KB increments. Although this case satisfies the alignment restrictions for Direct I/O, read-ahead will not occur when Direct I/O is used. Cached I/O again outperforms Direct I/O in this case due to file system read-ahead. Note that the total amount of data read in this case is the same for both Direct and cached I/O (although cached I/O reads one additional page, due to read-ahead).

The third row in Table 2 corresponds to the case where a 1GB file is read sequentially in 10MB increments. Direct I/O significantly outperforms cached I/O in this case for two reasons. First, the overhead of double-copying is eliminated with Direct I/O. Secondly, cached I/O does not see the benefit of read-ahead in this case because at most 8 4K pages can be read ahead (since **j2_maxPageReadAhead**=8), while the read increment in this case is 2560 4K pages.

These examples show that applications do not uniformly benefit from Direct I/O. However, applications that see performance benefits when using raw logical volumes for storage are likely to benefit from the use of Direct I/O. Raw logical volumes also impose alignment and length restrictions on I/O – they require that the application buffer be 512-byte aligned, and that lengths be in 512-byte increments. Thus, applications that use raw logical volumes for I/O already implement these alignment and length restrictions. By creating file systems with an appropriate block size (e.g., by specifying **agblksize**=512 at file system creation), such applications can benefit from the use of Direct I/O without any modification.

Table 2: Direct I/O vs. cached I/O performance

Read Increment	Total File Size	Cached I/O		Direct I/O	
		Elapsed Time (sec)	Total KB Read	Elapsed Time	Total KB Read
1 byte	1 MB	1.59	1,036	185.27	4,194,320
4 KB	1 GB	21.18	1,045,982	104.31	1,045,986
10 MB	1 GB	20.59	1,048,592	6.81	1,048,596

2.2 Inode Locking

While an application views a file as a contiguous stream of data, this is not actually how a file is stored on disk. In reality, a file is stored as a set of (possibly non-contiguous) blocks of data on disk. Each file has a data structure associated with it, called an *inode*.

The inode contains all the information necessary for a process to access the file, such as file ownership, access rights, file size, time of last access or modification, and the location of the file's data on disk. Since a file's data is spread across disk blocks, the inode contains a "table of contents" to help locate this data. It is important to note the distinction between changing the contents of an inode and changing the contents of a file. The contents of a file only change on a write operation. The contents of an inode change when the contents of the corresponding file change, or when its owner, permissions, or any of the other information that is maintained as part of the inode changes. Thus, changing the contents of a file automatically implies a change to the inode, whereas a change to the inode does not imply that the contents of the file have changed. Since multiple threads may attempt to change the contents of an inode simultaneously, this could result in an inconsistent state of the inode. In order to avoid such race conditions, the inode is protected by a lock, called the inode lock. This lock is used for any access that could result in a change to the contents of the inode, preventing other processes from accessing the inode while it is in a possibly inconsistent state.

When a file is accessed for reading, the contents of the inode do not change, whereas writes to a file do change the contents of the inode (and the contents of the file). JFS2 uses a read-shared, write-exclusive inode lock which allows multiple readers to access the file simultaneously, but requires that the lock be held in exclusive mode when a write access is made. This means that when the lock is held in write-exclusive mode by a process, no other process may access the file for either reads or writes. However, when the lock is held in read-shared mode by a process, other processes can concurrently read data from the file. Figure 6 depicts the serialization enforced by the inode lock in JFS2. In the figure, threads 1 and 2 simultaneously read data from a shared file. When thread 2 performs a write on the file, it takes the inode lock in write-exclusive mode, preventing thread 1 from performing reads or writes on the file for the duration that thread 2 holds the lock.

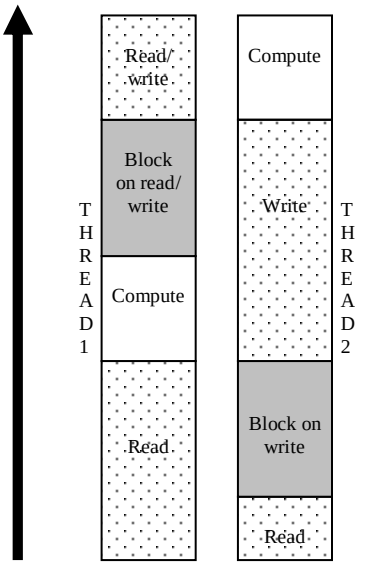

Figure 6: Read-shared, write-exclusive inode locking in JFS2

2.2.1 Concurrent I/O

The inode lock imposes write serialization at the file level. Serializing write accesses ensures that data inconsistencies due to overlapping writes do not occur. Serializing reads with respect to writes ensures that the application does not read stale data. Sophisticated database applications

implement their own data serialization, usually at a finer level of granularity than the file. Such applications implement serialization mechanisms at the application level to ensure that data inconsistencies do not occur, and that stale data is not read. Consequently, they do not need the file system to implement this serialization for them. The inode lock actually hinders performance in such cases, by unnecessarily serializing non-competing data accesses. For such applications, AIX 5L v5.2 ML01 offers the Concurrent I/O (CIO) option. Under Concurrent I/O, multiple threads can simultaneously perform reads and writes on a shared file. This option is intended primarily for relational database applications, most of which will operate under Concurrent I/O without any modification. Applications that do not enforce serialization for accesses to shared files should not use Concurrent I/O, as this could result in data corruption due to competing accesses.

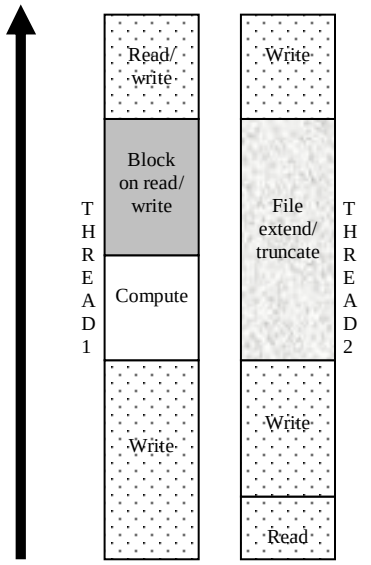

Figure 7: Inode serialization under Concurrent I/O on JFS2

2.2.1.1 Concurrent I/O Usage

Concurrent I/O can be specified for a file either through the mount command (**mount –o cio**), or via the **open()** system call (by using **O_CIO** as the *OFlag* parameter). When a file system is mounted with the **–o cio** option, all files in the file system use Concurrent I/O by default. Just as with Direct I/O, Concurrent I/O can be restricted to a subset of the files in the file system by placing the files that use Concurrent I/O in a separate subdirectory and using **namefs** to mount this subdirectory over the file system. For example, if a file system *somefs* contains some files that prefer to use Concurrent I/O and others that do not, we can create a subdirectory, *subsomefs* containing all the files that use Concurrent I/O. We can mount *somefs* without the **–o cio** option, and then mount *subsomefs* as a **namefs** file system with the **–o cio** option: **mount –v namefs –o cio /somefs/subsomefs /somefs**.

The use of Direct I/O is implicit with Concurrent I/O, and files that use Concurrent I/O automatically use the Direct I/O path. Thus, applications using Concurrent I/O are subject to the same alignment and length restrictions as Direct I/O, specified in **Table 1**. As with Direct I/O, if there are multiple outstanding opens to a file and one or more of the calls did not specify O_CIO, then Concurrent I/O is not enabled for the file. Once the last conflicting access is eliminated, the file begins to use Concurrent I/O. Since Concurrent I/O implicitly uses Direct I/O, it overrides the O_DIO flag for a file.

Under Concurrent I/O, the inode lock is acquired in read-shared mode for both read and write accesses. However, in situations where the contents of the inode may change for reasons other than a change to the contents of the file (writes), the inode lock is acquired in write-exclusive mode. One such situation occurs when a file is extended or truncated. Extending a file may require allocation of new disk blocks for the file, and consequently requires an update to the "table of contents" of the corresponding inode. In this case, the read-shared inode lock is upgraded to the write-exclusive mode for the duration of the extend operation. Similarly, when a file is truncated, allocated disk blocks might be freed and the inode's table of contents needs to be updated. Upon completion of the extend or truncate operation, the inode lock reverts to read-shared mode. This is a very powerful feature, since it allows files using Concurrent I/O to grow or shrink in a manner that is transparent to the application, without having to close or reopen files after a resize. Figure 7 shows the behavior of the inode lock under Concurrent I/O.

Another situation that results in the inode lock being acquired in write-exclusive mode is when an I/O request on the file violates the alignment or length restrictions of Direct I/O. Alignment violations result in normal cached I/O being used for the file, and the inode lock reverts to the

read-shared, write-exclusive mode of operation depicted in Figure 6.

2.2.1.2 Performance Considerations under Concurrent I/O

Since Concurrent I/O implicitly invokes Direct I/O, all the performance considerations for Direct I/O mentioned in Section 2.1.1.2 hold for Concurrent I/O as well. Thus, applications that benefit from file system read-ahead, or have a high file system buffer cache hit rate, would probably see their performance deteriorate with Concurrent I/O, just as it would with Direct I/O. Concurrent I/O will also provide no benefit for applications in which the vast majority of data accesses are reads. In such environments, read-shared, write-exclusive inode locking will already provide most of the benefits of Concurrent I/O.

Applications that use raw logical volumes for data storage don't encounter inode lock contention since they don't access files.

2.3 The Sync Daemon

The sync daemon (/usr/sbin/syncd) forces a write of dirty (modified) pages in the file buffer cache out to disk. By default, the sync daemon runs at 60-second intervals. On systems with large amounts of memory and large numbers of pages getting modified, this can result in high peaks of I/O activity when the sync daemon runs.

Since Direct I/O bypasses the file buffer cache and directly writes data to disk, the use of Direct I/O results in a reduction in the number of dirty pages that need to be flushed by the sync daemon. The same holds true for raw logical volumes.

3 Performance Test Environment

In order to evaluate Concurrent I/O performance, we measured the throughput of an online transaction processing (OLTP) workload under different database storage configurations. We used Oracle9*i* Database for this study. This workload uses a client/server model, where a client system drives the database server with a mix of transactions intended to simulate a user environment. Database throughput was measured in terms of the number of transactions completed per second (tps) at the client. The client also measured the response time characteristics of the transactions executed by the database server. We measured the performance of our workload under three different configurations for database storage:

- Raw logical volumes
- JFS2 filesystems with Direct I/O
- JFS2 filesystems with Concurrent I/O

The workload was update intensive, and consisted of a mix of transaction types on multiple tables. The system configuration used in these tests is specified in Table 3.

Table 3: System configuration

Attribute	Value
System Type	pSeries™ 680
Number of CPUs	4
Processor Type	RS64-IV
Processor Clock Speed	600 MHz
System Memory	48 GB
O.S. Level	AIX 5L version 5.2, 64-bit kernel
Database	Oracle9*i* Database release 2 v9.2.0.1 (64-bit)
Database size	500 GB
Configured Disks	>400 SSA

The OLTP workload consists of a ramp-up phase, followed by the steady-state phase, and ending with a brief ramp-down phase. On our measurement configuration, the steady state performance was reached within about five minutes of execution. The performance measurement interval lasts about thirty minutes after reaching steady state.

Database performance benchmarks typically use raw logical volumes for database storage. Raw logical volumes do not suffer from the overheads described in section 2 for file systems, and usually provide the best performance for database applications. We measured the performance of raw logical volumes to serve as the performance goal to be attained when database storage is done on file systems.

Table 4: Comparison of storage configurations

Raw LV configuration		JFS2 configuration	
Total no. of LVs	Total no. of disks	Total no. of files	Total no. of disks
193	420	51	420

3.1 Raw Logical Volume Configuration

The physical configuration for the OLTP run with raw logical volumes was designed to reduce I/O wait to as near to zero as possible. The objective was to maximize throughput and minimize I/O bottlenecks by spreading data across multiple logical volumes and disks. In all, 193 raw logical volumes were used in this configuration.

All of the raw logical volumes (LVs) were created in volume groups defined in 32MB partitions with the exception of the LVs for the database logs, which were defined in 128MB partitions. Previous exercises using this workload have shown that I/O disk performance is best when raw LVs are spanned (without striping) on the outer edge of the disks.

For the database logs, SSA RAID-5 arrays were created consisting of 4 physical disks (*pdisk*) for each logical disk (*hdisk*). The LVs and redo log files were 20GB in size, so as to eliminate log switches during the OLTP run. Unlike the other LVs, the LVs for the redo log files were striped across the two logical *hdisks*, with a stripe size of 128K.

3.2 JFS2 File System Configuration

As previously mentioned, Direct I/O (DIO) eliminates the overheads associated with the file buffer cache. However, the write-exclusive inode lock continues to pose a performance bottleneck with Direct I/O. In order to minimize the effects of spinning on the inode lock, file system based database applications tend to maximize the number of files in their configuration in order to increase the number of concurrent writes in progress. In previous experiments involving JFS2 file systems with the Direct I/O option, we usually created as many file systems as the number of raw logical volumes used in the raw LV configuration. However, using a large number of files increases the complexity of database administration. A smaller number of files results in a more easily manageable configuration from a database administrator's perspective.

For this study, we chose to limit the number of files to a manageable number, rather than matching the number of raw logical volumes. This choice eliminated a direct one to one performance comparison between raw and JFS2 physical layouts and implied a potential performance loss due to the use of very large data files. In addition, we treaded on performance waters by striping the JFS2 volumes on raw devices, as opposed to the traditionally preferred method of spanning volumes on raw devices. However, as the very positive JFS2 results show, as long as the number of disks remains the same, DBAs can consolidate raw LVs into JFS2 LVs without risking loss in performance.

For the Concurrent I/O run, the filesystems were mounted with the **–o cio** mount option, and for the Direct I/O run, they were mounted with the **–o dio** option. We used a file system block size (*agblksize*) of 4KB. In general, the file system block size used should match the database block size, in order to satisfy Direct I/O alignment restrictions. For the measurements reported in this section, the database transaction logs (redo logs) were stored on raw logical volumes. Using file systems for the redo logs, with *agblksize*=512 bytes, resulted in about a 4% reduction in throughput. Using *agblksize*>512 bytes results in much worse performance, as this violates the Direct I/O alignment restrictions that need to be satisfied in order for Concurrent I/O to be used for the file. Database control and configuration files, which are frequently accessed but do not usually satisfy Direct I/O alignment constraints, would exhibit better performance under cached I/O rather than Direct or Concurrent I/O.

The thread scheduling policy used in our experiments was the default SCHED_OTHER policy.

The OLTP workload measured in our experiments used asynchronous I/O for writing database buffers to disk. Asynchronous I/O (AIO) in JFS2 is handled by kernel processes (**kprocs**), called **aioservers**. AIO requests from the application are queued into an AIO queue. An

aioserver picks requests off the AIO queue one at a time, and is unable to process any more requests until I/O has completed for the request it is currently servicing.

Thus, the number of **aioservers** in the system limits the number of asynchronous I/O operations that can be in progress simultaneously. The maximum number of **aioservers** that can be created is controlled by the **maxservers** attribute, which has a default value of 10 per processor. For our experiments, we used a **maxservers** value of 400 per processor.

4 Performance Test Results

In this section, we compare various performance metrics for each of the three storage configurations measured. For ease of comparison, this data is presented through a series of graphs. For each graph, we present relevant observations to help explain the depicted behavior. In the graphs presented in this section, DIO stands for Direct I/O, and CIO for Concurrent I/O.

4.1 Throughput

Table 5 lists the average throughput and response times for each storage configuration, measured over the steady-state interval of each benchmark run. While the database throughput with Direct I/O was 70% lower than with raw LVs, the throughput with Concurrent I/O was only 8% lower than the raw LV case. As our graphs will corroborate, the poor Direct I/O performance can be largely attributed to severe lock contention for the inode locks. The Direct I/O performance could have been improved by using a larger number of files, such as by creating one file per disk, as this would have reduced inode lock contention.

Figure 8 plots the throughput measured at 30-second intervals over the duration of the run. The two dotted vertical lines demarcate the ramp-up, steady-state and ramp-down phases of the runs. The difference in performance between Concurrent I/O and raw LVs stems from two factors. The first is simply the additional pathlength due to I/O requests having to go through the file system path. The second factor has to do with the way asynchronous I/O is handled through a file system, as opposed to when raw LVs are used. Since the AIO server threads are kernel threads, a context switch occurs for each AIO request to a file. On the other hand, AIO requests to raw logical volumes use a "fast path" to the logical volume manager that avoids the use of AIO server threads.

Table 5: Average throughput and response times

Configuration	Avg Throughput (tps)	Avg Response Time (sec)
Raw LVs	710.19	0.06
Direct I/O	219.69	0.12
Concurrent I/O	652.12	0.07

Apart from this difference in performance, the behavior of the Concurrent I/O run is remarkably similar to the raw LV run. Both exhibit very low variation in throughput once steady-state is reached. The slight variation in throughput is merely because of normal load variations in the benchmark. The Direct I/O run displays wide variations in throughput, which is symptomatic of severe lock contention.

4.2 Disk I/O

Figure 9 shows the disk I/O rates for each of our runs. The I/O rate stays fairly constant throughout the run, and mirrors the throughputs presented in Figure 8. This is expected, since the amount of I/O per transaction is constant, so a lower throughput number would also correspond to a lower I/O rate. It is interesting to note that file system journaling does not appear to increase the disk I/O rate in the file system runs. This is because of the OLTP workload behavior. Files are created and populated when the database is created. The run only uses pre-initialized files, and there is virtually no file creation, deletion, allocation or truncation during the run. Thus, there is little journaling during the runs.

The disk farm used in our experimental setup did not pose a bottleneck in any of our runs. Figure 10 plots the highest disk utilization for any disk within each 30-second measurement interval, over the duration of the benchmark run. As shown in the graph, the highest disk utilization by any disk at any point did not exceed 70% during any of our runs. CPU usage statistics showed no I/O wait times in any of our runs.

4.3 CPU Usage

CPU usage statistics are a commonly used metric for discussing system performance. We used the AIX *vmstat* command to collect CPU usage statistics at 30-second intervals for the entire

duration of each run. The **vmstat** command breaks down CPU usage into four components: *%user*, *%system*, *%idle*, and *%wait*. *%user* is the percentage of CPU time spent executing user-level instructions. *%system* is the percentage of CPU time spent executing at the system (supervisor) level. *%idle* is the percentage of time that the CPU is available for additional work. *%wait* is the percentage of time that the CPU spends waiting for I/O to complete, with no runnable processes.

Figures 11, 12 and 13 respectively show the *%user*, *%system* and *%idle* times for each run, as reported by **vmstat**. None of our benchmark runs displayed any %wait time. The high CPU utilization (*%user* + *%system* ≈ 100%) is typical of the OLTP benchmark, as there is little or no disk I/O wait.

The Concurrent I/O run has a higher *%system* and lower *%user* than the raw LV run because of the additional context switches due to traversing down the file system path for I/O, and due to the use of AIO server threads for servicing AIO requests. This results in a greater amount of work being done at the system level. As such, the OLTP workload is not system-intensive. Neither the Concurrent I/O nor the raw LV runs show any idle time, whereas the Direct I/O run shows a high percentage of idle time. This, again, is indicative of severe lock contention in the Direct I/O run.

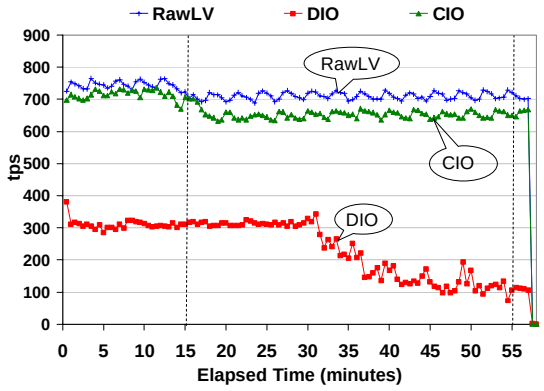

Figure 8: Benchmark throughput over run duration. Higher *tps* indicates better performance.

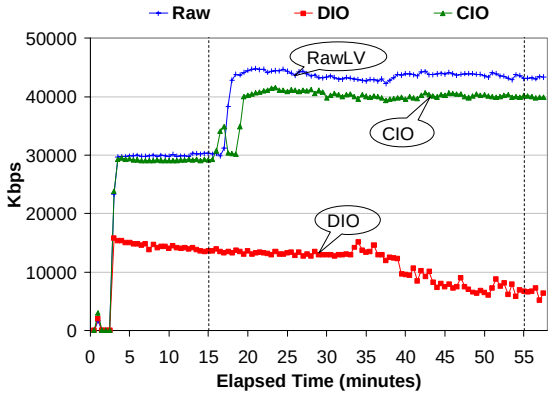

Figure 9: Disk I/O throughput (in kilobytes of data transferred per second) over run duration.

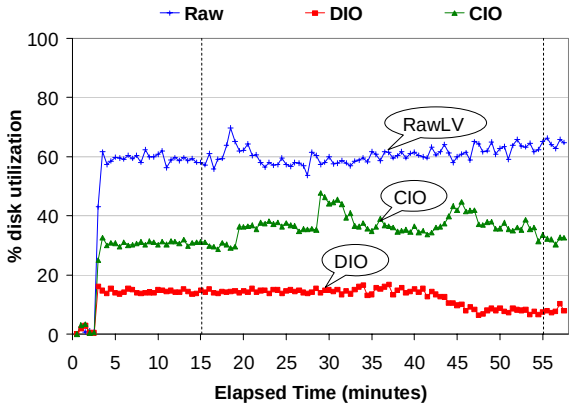

Figure 10: Highest disk utilization over run duration.

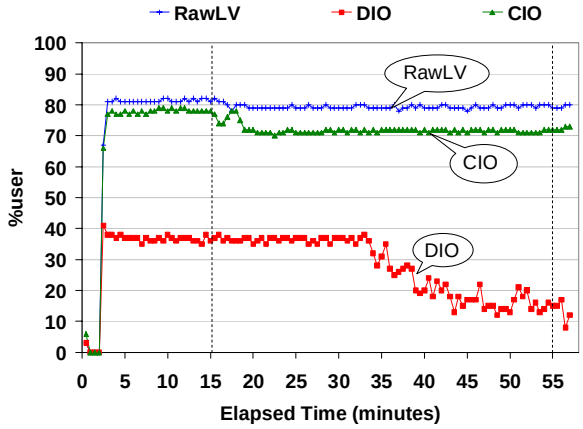

Figure 11: %user over run duration

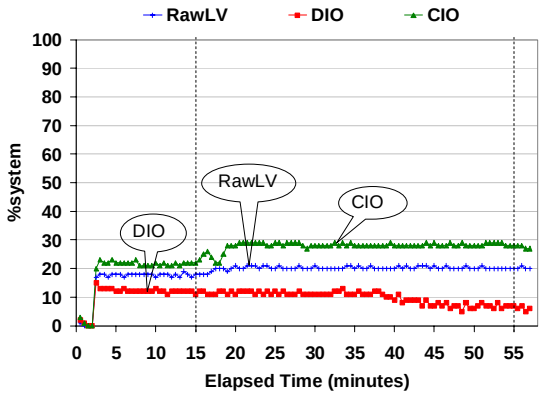

Figure 12: %system over run duration

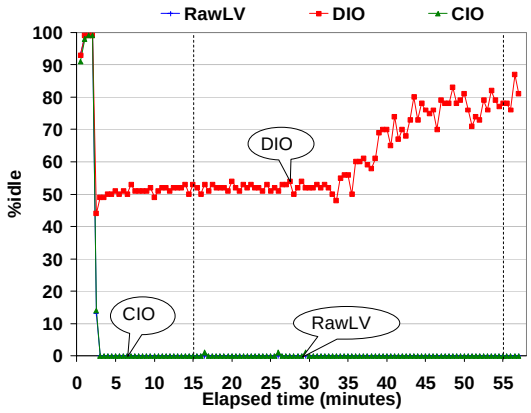

Figure 13: %idle over run duration.

4.4 Lock Statistics

The study of the locking behavior exhibited by the three storage models reveals the most information about their performance characteristics. The lock statistics reported here were gathered using the AIX *trace* facility. The traces were gathered over a five-second interval during the steady state phase of each run.

To better understand the locking statistics presented here, a brief explanation of the AIX 5L v5.2 locking process is in order. When a thread first attempts to acquire a lock and fails, it may spin around the lock for at most *maxspin* number of times. The variable *maxspin* can be set via the AIX **schedo** command. For our runs, we used the default *maxspin* value of 16384. If the thread fails to acquire the lock after *maxspin* attempts, it goes into wait state and is undispatched. When the lock is released by the owning thread, it wakes up one or more of the threads waiting to acquire that lock, and the cycle repeats itself. Heavy lock contention thus manifests itself as through a large number of threads going into the wait state, or *blocking*, while trying to acquire the lock.

Figure 14 plots the number of blocks per second for the most contended lock classes in our tests, i.e., the number of times any thread was driven into wait state while waiting for a lock during the measurement interval, divided by the measurement interval. By "lock class", we mean all the locks that fall under the same functional category. For example, the lock class "SSA" includes all the different locks used for serializing accesses to SSA devices.

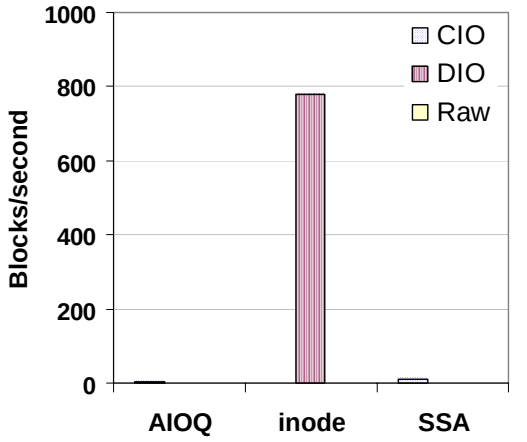

Figure 14: Blocks/second for major lock classes

The three major lock classes observed in our traces were:

- **AIOQ lock** – serializes insertion and removal of AIO requests from the AIO queues.
- **Inode lock** – the per-file write-exclusive lock used by the file system to serialize write accesses to a file. This lock is not taken in Concurrent I/O, except under the conditions listed in Section 2.2.1.1.
- **SSA lock** – used for serializing accesses to an SSA device. Each device has its own lock.

As seen in Figure 14, the Direct I/O run shows excessive amounts of blocking due to the inode lock, which results in a lot of context switching. Since a large number of threads went into the wait state in the Direct I/O run, there was a significant amount of CPU idle time when there were no runnable threads, resulting in lower

throughput. The Concurrent I/O and raw LV runs do not acquire write-exclusive inode locks.

5 Conclusion

The performance experiments described in this paper show that using JFS2 Concurrent I/O for databases results in performance comparable to that achieved through the use of raw logical volumes for database storage, while providing greater flexibility and ease of administration. The throughput of the database application under Concurrent I/O was three times the throughput achieved with Direct I/O – a 200% improvement - and lagged the performance on raw logical volumes by only 8%. We have also shown that the performance equivalence between Concurrent I/O and raw logical volumes holds at a detailed level, with many system metrics showing remarkably similar behavior in both cases.

Thus, Concurrent I/O combines all the performance advantages of using raw logical volumes, while greatly simplifying the task of database administration. This makes Concurrent I/O a very attractive option for database storage.

IBM Corporation
Marketing Communications
Server Group
Route 100
Somers, New York 10589

Produced in the United States of America
05-03