# High Performance I/O

By Bret R. Olszewski and David B. Whitworth

*The challenge of supplying high-performance I/O to applications is never ending. AIX has provided services such as I/O coalescing, read-ahead/write-behind algorithms, and software striping to maximize device throughput. These algorithms frequently trade processor cycles (CPU time) for device throughput. Further optimization, improving CPU cycles for I/O, is possible in some circumstances with the direct I/O feature of AIX 4.3.*

To understand direct I/O, it is necessary to review the basics of AIX® disk I/O. Since AIX is a general-purpose operating system, it supports a variety of mechanisms for disk I/O. Figure 1 shows the component stack for three cases of disk I/O: file system, memory-mapped I/O, and raw I/O.

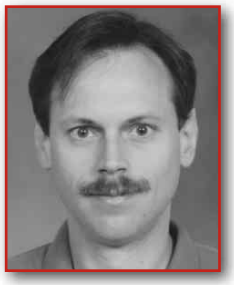
Starting from the bottom of the stacks, the layer closest to the physical disk device is the device driver. The device driver understands the characteristics of the physical device and how to make it perform primitive read, write, and status operations. On top of the device drivers is the logical volume manager (LVM). The LVM layer is rich in function. Some of its capabilities include logical volumes that span multiple physical disks, mirroring, and error recovery for physical disk problems. Above the LVM are the file systems.

AIX supports several file systems, including the journaled file system (JFS), network file system (NFS), distributed file system (DFS), CD-ROM file system (CDRFS), the special file system (SPECFS), as well as others. This article will concentrate on the JFS and the SPECFS, although many of the principals of these file systems apply to others as well. Above the file systems is the logical file system (LFS). This layer makes it possible for user applications to do I/O to different file systems using a single set of program interfaces. Each disk I/O involves a trip through the appropriate layers of software. The path length, or number of instructions required for the disk I/O, is the sum of the instructions executed in each layer. Figure 1 shows the disk I/O paths.

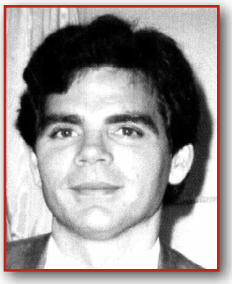
File systems typically support the following basic functions:

- **Consistency.** If a file is being read and/or written by multiple threads, each thread gets a consistent view of the data. The file system serializes access to the physical file data. This primitive consistency model is usually augmented by application locks for sophisticated applications.

- **Buffering.** Application programs typically have file access patterns that are inefficient with regard to the physical disk devices. When an application opens a file and performs reads from it, the application reads can involve disk I/O or they can be resolved with buffered data. For example, a program doing 4096 sequential one-byte read



Bret R. Olszewski



David B. Whitworth

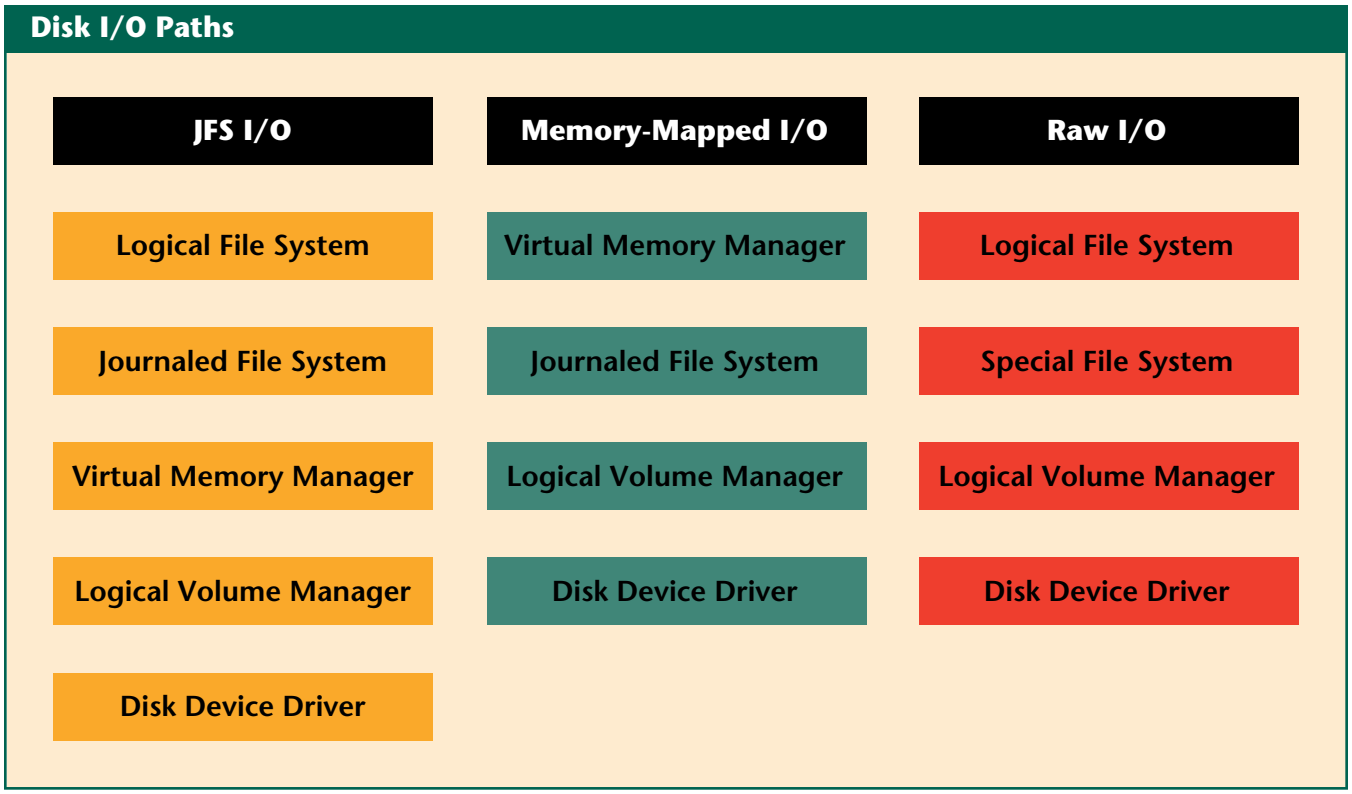



*Figure 1. Disk I/O Paths*

system calls will typically only need a single 4096 disk read. When data is read, a copy of the data is normally kept in memory, or buffered, as a function of AIX's virtual memory manager. If a file is frequently accessed, the file data may remain in system memory so that physical disk I/O is not needed.

- **Allocation.** A file system manages space. As files are extended, the file system allocates disk blocks. When files are truncated or deleted, the file system takes back disk blocks as free space.

- **Resource management.** A file system can manage resources such as buffer management and space quotas. It can limit how much buffering is allowed for any particular file, based on the amount of memory available as well as file usage pattern. It can limit how many disk blocks may be allocated to any user.

File system I/O via JFS is performed through system calls such as `read` and `write`. These system calls manage the file data and deal with buffering. The buffering of file data is contained within the kernel address space. This means that the file data cannot be directly accessed by user programs. Data is moved to or from user space via the system calls (that is, `read` and `write`).

Memory-mapped I/O is a means of mapping files directly into program address space (available on AIX via the `shmat` and `mmap` system calls). In the case of memory-mapped I/O, the file data is accessed by memory location (that is, not via `read/write` system calls) and no copying of data is needed. The granularity of access is a 4096 byte page. Because reads and writes are handled transparently by the virtual memory manager, applications that share files for updating must provide their own serialization, typically via mutex locks, to ensure consistency of data.

Since the underlying file system supports allocation and resource management, raw I/O bypasses the traditional file system completely. On AIX it typically passes through a thin veneer of function in the SPECFS. Essentially, it skips file system function completely, not providing any consistency, buffering, allocation, or resource management services. Because of its limited service, applications that have multiple threads updating data must provide serialization. The application also must manage allocation of space within the logical volume. Since there is no buffering, stricter alignment rules apply to data and correctly aligned buffers require no copying. Raw I/O is typically the most efficient in path length—that is why it is normally used for large database benchmarks such as TPC-C and TPC-D.

An alternative to kernel caching would be to have the file contents cached in user addressable memory. User caching has potential drawbacks. Consider what happens if a thread accidently modifies file storage, perhaps by an errant pointer reference. The accident will not be caught and the file data may be corrupted. Other hardware-related issues, such as the need to use instructions that are only legal in kernel mode to change addressability as well as address space limitations in 32-bit microprocessors, limit the effectiveness of user space caching with full UNIX® semantics. For many applications, AIX's support of user addressable caching via the `shmat` and `mmap` system calls is efficient.

## Direct I/O Implementation

Most users prefer the convenience of using file systems, but some require greater performance. A side effect of the continual increase in microprocessor performance has been that memory bandwidth has not always kept up. To put it bluntly, the speed that data can be copied from one memory location to another is dependent on memory subsystem performance and is relatively independent of microprocessor speed. This manifests itself in several ways, one of which is that block copy operations become a significant percentage of the execution time for some I/Os. In fact, it is possible on some systems to achieve I/O bandwidths that are limited by the amount of system processing power—not the physical disk devices. To mitigate this effect, direct I/O merges some of the characteristics of file system I/O and raw I/O.

***Programs that are typically CPU-limited and do lots of disk I/O are good candidates for direct I/O.***

There is no such thing as a free lunch. Direct I/O imposes a number of restrictions on programs. Because direct I/O is uncached, as is raw I/O, no copy operation is required. The flip side of this is that there is no buffering, so two reads of the same data will result in two disk I/Os. Also, the alignment of the user buffer must be compatible with the file allocation. The alignment requirement is due to the fact that the disk data is moved into the user buffer by direct memory access (DMA) from the disk device.

Direct I/O does not benefit from VMM `read ahead` or `write behind`, so it may be necessary to increase the size of the I/Os an application does to match the sequential throughput the application gets with normal I/O. I/Os of 32 KB to 128 KB or larger should provide good throughput.

At present, direct I/O is only supported for program working storage. Direct I/O will not work to client segments (that is, files mapped to file systems such as NFS or DFS) or to shared memory segments mapped with `mmap`. Figure 2 summarizes the capabilities and restrictions of the various I/O mechanisms.

So what types of programs are good candidates for direct I/O? Programs that are typically CPU-limited and do lots of disk I/O. In particular, "technical" codes that have large sequential I/Os are good candidates. Applications that do numerous small I/Os will typically see less performance, because

**Comparison of Function**

| I/O Mechanism | F/S R/W | Memory Mapped | Direct | Raw |
|---|---|---|---|---|
| Alignment required | No | Yes | Yes | Yes |
| Buffered | Yes | Yes | No | No |
| Read/write consistency | Yes | Yes | Yes | Yes |
| Read ahead | Yes | Yes | No | No |
| Write behind | Yes | Yes | No | No |
| Copy required | Yes | No | No | No |
| Serialization (`fcntl`, `flock`, `lockf`) | Yes | No | Yes | No |
| Client segments | Yes | NFS | No | No |
| Mmap segments | Yes | N/A | Yes | No |
| Compression | Yes | Yes | No | No |
| Striping | Yes | Yes | Yes | Yes |
| AIO support | Yes | Yes | Yes | No |

*Figure 2. Comparison of function*

direct I/O cannot do `read ahead` or `write behind`. Applications that have benefited from striping are also good candidates.

## Performance Benchmarks

To compare the amount of CPU consumed for raw I/O, direct I/O, and traditional file system I/Os, we use a set of disk performance benchmarks. The performance is measured by a program that does either sequential or random reads from varying numbers of disks. The sustained throughput and CPU utilization are reported. Our tests are performed with multiple disks and adapters. Your performance will vary, depending on system configuration.

The JFS and Direct tests read one JFS file per disk and the Raw test reads one logical volume per disk. The random tests do 4 KB I/Os. The JFS and Direct I/O sequential tests do 32 KB I/Os and the Raw sequential test does 128 KB I/Os. The Raw sequential test has a slight advantage because it does larger I/Os. The benchmark ensures that none of the data to be read is cached so that all reads come from the disks.

Results are included for the two different systems: the 12-processor Model S70 symmetric multiprocessor (SMP) and the uniprocessor E30. The two systems provide a good comparison because of the difference in the relative memory bandwidth available. Figure 3 shows the CPU utilization of the E30 system for varying levels of disk throughput doing sequential reads.

Sequential reads allow maximum disk bandwidth because disk seeks are avoided. Notice that reading files through standard JFS saturates the CPU at less than 20 megabytes per second, while direct I/O and raw I/O are consuming 15% or less of the CPU at the same throughput. Also notice that raw I/O is still more efficient than direct I/O, but they have much closer performance compared with standard JFS I/O.

Figure 4 shows the CPU utilization of the S70 system on the same benchmark. Since the memory bandwidth of the system is much higher than the E30, we were unable to saturate the CPU of the S70 with disks available for this benchmark. Notice that, even though the S70 has much more memory bandwidth, the shapes of the curves are similar to those on the E30. This indicates a

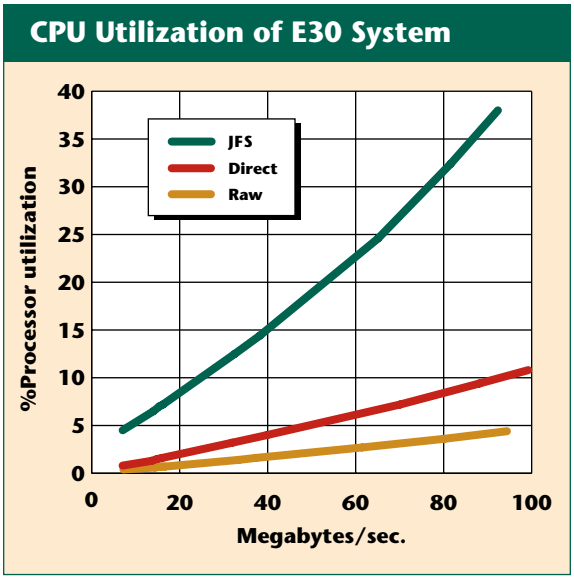



*Figure 3. Disk throughput for sequential reads*

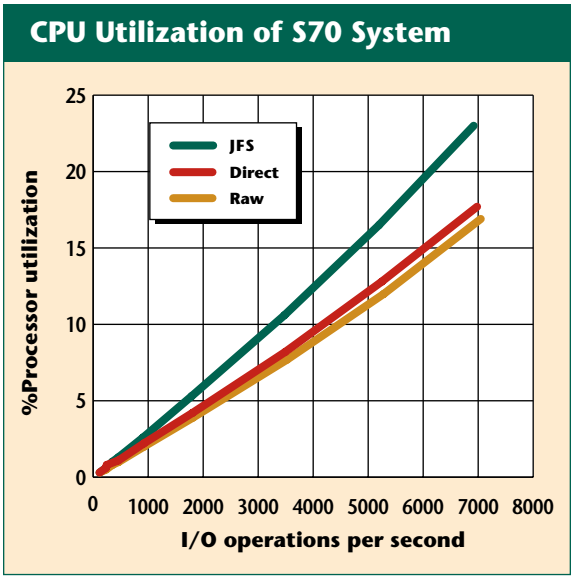



*Figure 4. Disk throughput for sequential reads*

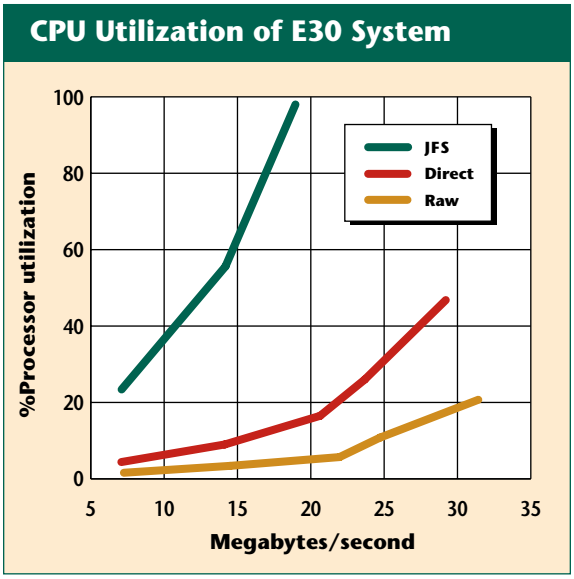



*Figure 5. Random I/O*

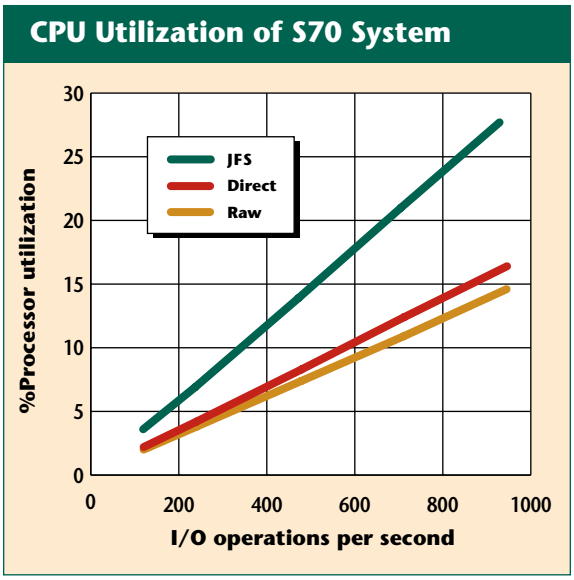



*Figure 6. Random I/O*

similar level of benefit of direct I/O on systems with high memory bandwidth.

Figures 5 and 6 show the CPU utilization of the E30 and S70 systems for the random I/O part of the benchmark. Since the random I/Os are smaller (4096 bytes instead of 32 kilobytes), the copy overhead of the I/O is less for the random test than for the sequential test. Put another way, a single, large I/O has efficiencies in path length compared to several small I/Os, even if the total amount of read or written (and copied in the standard JFS case) is the same. In both cases, the benefit of direct I/O is substantial, though considerably less than in the sequential cases.

## Using Direct I/O

The use of direct I/O is fairly simple; none the less, we recommend consulting standard AIX documentation before modifying applications. One thing to consider is that

direct I/O is a function of JFS and is not supported on most other file systems. For example, trying to open a file mounted over NFS for direct I/O will function correctly, but not benefit from direct I/O performance. AIX provides the `ffinfo` system call to query the required alignment, minimum read/write size, and maximum read/write size of the file using direct I/O. The `ffinfo` system call only returns meaningful information when used on JFS files. Direct I/O is enabled via the `O_DIRECT` flag on the `open` system call.

Figure 7 shows a simple program that does a single 4 KB read via direct I/O. The key points are include files, the open option, buffer alignment, and the `ffinfo` call. The include file `fcntl.h` is needed for the open option. The include file `sys/finfo.h` is needed for the `finfo` system call. On the open, `O_DIRECT` is used to inform the file system that direct I/O is desired. The buffer is allocated from the heap via `malloc`. However, `malloc` does not guarantee alignment, so the program `malloc` has twice as much space as needed and uses an aligned buffer

```
#include <fcntl.h>
#include <sys/finfo.h>

main (argc, argv)
int     argc;
char    *argv[];
{
#define BUFFER_SIZE 16384
#define ALIGN 4096

        char    errmsg[80];
        char    *filename;
        int     fd;
        char    *buffer;
        struct diocapbuf buf;

        /* 4 KB align the I/O buffer */
        buffer = (char *) malloc(BUFFER_SIZE+ALIGN);
        buffer = (char *)(((int)buffer + ALIGN) & 0xfffff000);

        if ( argc < 2) {
                printf("usage: %s <data file>\n", argv[0]);
                exit ( 1 );
        }
        filename = &argv[1][0];

        if ( ( fd = open ( filename, O_DIRECT | O_RDONLY ) ) < 0 ) {
                sprintf (errmsg, "%s: cannot open %s", argv[0], filename);
                perror (errmsg);
                exit (1);
        }

        if ( ffinfo ( fd, FI_DIOCAP, &buf, sizeof(buf) ) < 0 ) {
                sprintf (errmsg, "%s: ffinfo failed %s", argv[0], filename);
                perror (errmsg);
                exit (1);
        }
        printf ( "buf.dio_offset = %d\n", buf.dio_offset );
        printf ( "buf.dio_max = %d\n", buf.dio_max );
```

*(continued on next page)*

*Figure 7. Example of single 4 KB read via direct I/O*

*(continued from previous page)*

```
        printf ( "buf.dio_min = %d\n", buf.dio_min );
        printf ( "buf.dio_align = %d\n", buf.dio_align );
         if (buf.dio_align != ALIGN)
              printf("alignment does not allow direct I/O!\n"):


        if ( read ( fd, buffer, BUFFER_SIZE ) < 0 ) {
                sprintf ( errmsg, "%s: bad read", argv[0]);
                perror ( errmsg );
                exit ( 1 );
        }

        close ( fd );
        exit ( 0 );
}
```

***Figure 7. Example of single 4 KB read via direct I/O***

within the `malloc` allocation. The `finfo` call gets attributes about direct I/O support for the file.

## Conclusion

Direct I/O requires substantially fewer CPU cycles than regular I/O on today's computers. I/O-intensive applications that do not get much benefit from the caching provided by regular I/O can improve performance by using direct I/O. The benefits of direct I/O will grow in the future as increases in CPU speeds continue to outpace increases in memory speeds.

***Bret R. Olszewski**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Olszewski is a senior programmer working on MP performance. He joined IBM in 1989 and has worked on various aspects of AIX performance. He has a BS in Computer Science from the University of Minnesota.*

***David B. Whitworth**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Whitworth is a staff programmer in AIX Performance. He has a BA in Computer Science from the University of Texas.*