SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 26(9), 987–997 (SEPTEMBER 1996)

Comparative Analysis of Asynchronous I/O in Multithreaded UNIX*

H. CHUCK YOO
Korea University, Dept. of Computer Science, 1, 5-Ka, Anam-dong, Sungbuk-ku, Seoul 136-701, Korea

SUMMARY

I/O operations in UNIX are inherently synchronous. The need for asynchronous I/O comes first from multithreaded applications where thrads cannot block for I/O, and second from the fact that asynchronous I/O has much less overhead than synchronous I/O. There are two main approaches to accomplishing asynchronous I/O in UNIX. We compare the two approaches in design and implementation, and report the results of extensive experiments to measure the performance differences.

KEY WORDS: multithreading; blocking I/O; asynchronous I/O; UNIX; DDI/DKI; threads

INTRODUCTION

In UNIX, read() and write() calls to both block and raw devices are synchronous, i.e., the calling process sleeps inside the kernel after issuing an I/O request. When the I/O operation for the request completes, the interrupt handler wakes up the process, and the process returns with a proper error code. However, due to the buffer cache, read() and write() calls to file systems may not put the calling process to sleep. The buffer cache contains the data in recently-used disk blocks. When the buffer cache hits, read() returns with the buffer without waiting for the data, and write() just copies the data to the buffer cache and returns. However, the buffer cache exists only for file systems and therefore I/O for raw devices must be synchronous.

Asynchronous I/O means that after issuing an I/O request, the calling process returns immediately without sleeping. It checks the completion of the I/O request later. Asynchronous I/O may look similar to non-blocking I/O because, in both cases, the caller does not sleep. However, their semantics are very different. Non-blocking I/O returns an error if the caller would have to sleep. In other words, non-blocking I/O tries to use the data in the buffer cache, and if the data is not in the buffer cache, it does not initiate an I/O operation to do actual read or write but just returns. Asynchronous I/O, on the other hand, actually issues an I/O request but does not wait for completion.

The need for asynchronous I/O arose from two circumstances. One is in multithre-

* UNIX is a trademark of X/Open Company Ltd.

CCC 0038–0644/96/090987–11

Received 1 August 1995
Revised 12 January 1996

aded applications where multiple threads run in an address space. When a thread issues an I/O request, it cannot block for the request to be completed* because blocking stops the execution of all threads in the address space. What the thread needs to do is to issue an asynchronous I/O request and pass control to other runnable threads in order to continue execution of the application. Thus asynchronous I/O is a requirement for multithreaded applications.

The other circumstance is the efficient support of heavy I/O workloads. Large system environments such as decision support systems are often characterized by very heavy I/O loads. The overhead of synchronous I/O in such environments reduces the overall throughput significantly. An example of the overhead is context switching that takes place when a process blocks. When thousands of I/O requests are outstanding for a second, the context switching overhead is not negligible. Our previous paper[1] showed that asynchronous I/O has much less overhead than synchronous I/O, and can therefore support heavy I/O workloads more efficiently. New applications such as video-on-demand are also expected to involve heavy I/O workload, so asynchronous I/O is a crucial component to support such applications.

Although asynchronous I/O can be designed in various ways, there are two main approaches. One is to use a library to handle the asynchronous I/O requests. We call this the *library* approach. The other approach is to handle the requests in the file system layer of the kernel, called here the *kernel* approach. The goal of this paper is to provide an in-depth comparison of these two approaches.

BACKGROUND AND RELATED WORK

Background

The UNIX I/O subsystem consists of file system and device driver layers.[2] The file system layer handles file-related operations such as read() and write(), and translates them into device-specific operations. The device driver layer executes the device-specific operations passed from the file system layer. So device drivers are highly machine-dependent and device-dependent, whereas the file system layer is relatively machine-independent.

This paper assumes that the interaction between the file system layer and device drivers follows the System V Release 4 (SVR4) device driver interface/device kernel interface (DDI/DKI) framework. The SVR4 UNIX represents the majority of contemporary UNIX derivatives because SVR4 combines Berkeley UNIX and System V UNIX. In SVR4, the interface between the file system layer and device drivers is well defined as DDI/DKI.[3] In the DDI/DKI framework, a synchronous read or write call invokes the device driver's read or write entry point. The driver entry points call physio() which invokes a driver strategy routine to transfer the data. Then physio() calls biowait() to block the calling process. Later when I/O is done, the interrupt handler invokes biodone() to wake up the caller.

The fundamental abstraction that the UNIX I/O subsystem provides is files. In fact, every device in the UNIX kernel is accessed through a file interface. The

* If the kernel supports more than one control of execution in an address space, such as light-weight processes (LWP) in Sun's Solaris 2 operating system, a thread can sleep without affecting others if the thread is bound to an LWP. Threads will be explained in detail in a later section.

devices are divided into two categories: block and character devices. The block devices are ones with random access capability such as disks, and are accessed through block special files. The character devices are all non-block devices and are accessed through character special files. A block device may also be accessed through a character special file.

Block devices allow one to build file systems that support regular files, in which information can be stored and retrieved without knowing the details of the devices and the actual location where information is stored on the device. The most common example is a file system on a disk. The UNIX kernel is responsible for keeping track of allocation and de-allocation of individual disk blocks for files. File systems also maintain a buffer cache of recently-used disk blocks in order to minimize the frequency of disk accesses. The buffer cache is enhanced with read-ahead and delayed-write operations. The read-ahead is to guess what would be the next block needed and to request that block when the kernel reads a block from an I/O device. This enhances the performance of sequential reads. The delayed-write operation delays the write operation as late as possible so that perhaps the buffer will be reused before the write operation actually takes place. The dirty buffers in the buffer cache are pushed to disks periodically by a daemon process, called pageout.

On the other hand, character (or raw) devices do not support regular files, and the file system layer just passes the operations to a proper device driver after it does simple file-related operations such as permission checking. The focus of asynchronous I/O has been on raw devices because an I/O request to a file system is rendered asynchronous by the buffer cache.

There are several application programming interfaces (APIs) for asynchronous I/O. But this paper does not assume any particular one. Instead we use the generic term asynchronous read and write because every asynchronous API has the equivalent ones. After issuing asynchronous reads or writes, there are three ways that the caller can be notified. First, the caller can poll at its convenience to see if the request is done. The second notification method is that the caller receives a signal when an I/O request is completed. The third method is to watch a specified variable. For example, POSIX[4] sets the error status to EINPROGRESS until an asynchronous I/O request is done. So the change in the value indicates the I/O completion. An application can choose any of these notification methods.

Related work

Asynchronous I/O has existed in various forms in many systems. In non-UNIX systems, Digital Equipment Corporation's VMS operating system[5] used QIO (queued I/O) which is similar to asynchronous I/O. Several UNIX vendors have provided asynchronous I/O facilities. For example, Pyramid Technology Corporation has implemented asynchronous I/O in their OSx operating system,[6] and the API is an ioctl() interface. An implementation on IBM AIX/370 is described in Reference 7, and the API is based on the POSIX interface. That paper contains only the design issues and no experimental results are shown. Others such as Cray Research also provide an asynchronous I/O facility. But there have been very few reports discussing reasonable details of asynchronous I/O in these systems. There has not been any data confirming the advantage of asynchronous I/O over synchronous I/O in real applications until Reference 1.

In our previous study[1] we investigated asynchronous I/O as a high performance alternative to synchronous I/O. The environment under consideration was that the cooperating processes work together in parallel but they are divided into two classes: compute processes and I/O processes. Compute processes pass the data to I/O processes when I/O needs to be done. The I/O processes handle the buffers that contain the data and issue I/O calls. In other words, I/O processes are responsible for bringing in the necessary data to the buffers and for pushing the dirty buffers to disks. So the compute processes are decoupled from the I/O overhead. The purpose of decoupling is to achieve high I/O throughput. Of course, I/O processes and compute processes have to synchronize with each other.

We observed that when I/O processes use synchronous I/O, the number of I/O processes had to increase in order to keep up with the increase in the number of I/O requests. On the other hand, when I/O processes use asynchronous I/O, just one I/O process was able to handle the various workloads. We concluded that asynchronous I/O has indeed a performance advantage over synchronous I/O.

LIBRARY APPROACH

The basic concept in the library-based asynchronous I/O is to maintain queues of I/O requests in a library. When an application issues an asynchronous I/O request, the request is queued inside the library. Then the caller returns and continues its own execution. In parallel with the caller, the library picks up the requests and issues synchronous I/O calls for the requests. When the I/O calls complete, the library notifies the caller. In this way, the library provides an asynchronous view to applications but the library itself uses traditional synchronous I/O calls. In order to issue the I/O calls in parallel with the application, the library needs execution entity. Threads are the execution entity well suited for the library-based asynchronous I/O.

Because the term 'thread' has been used in different ways in the literature, we first explain what we mean here by thread. Traditionally a process has one execution entity so that there is one execution point in the address space of the process. Thread is an abstraction that allows multiple execution points in an address space. There are two types of threads: kernel-level threads and user-level threads. Kernel-level threads are threads in Mach[8] and light-weight processes[9] in Sun's Solaris 2 operating system. They are threads that run in a user address space but are known to the kernel. In other words, their data structures are allocated inside the kernel, and they are separate execution entities scheduled by the kernel. They behave much like processes except that they share a single address space so they are 'lighter' than traditional processes.

User-level threads provide multiple execution points in a user's view. The kernel may not, however, know of the existence of user-level threads. For example, user-level threads can switch themselves within a user process without involving the kernel. By 'multithreaded applications' we mean applications that use user-level threads in themselves. If a kernel supports both kernel- and user-level threads, the user-level threads can multiplex over the pool of kernel-level threads to maximize the benefits of the two types of threads. For further details of threads, please refer to the literature such as Reference 10.

What the library approach needs is kernel-level threads because kernel-level threads are separate execution entities that can run in parallel without causing the application

to block. Figure 1 illustrates the library approach. The queued I/O requests can be picked by kernel-level threads one at a time. The kernel-level threads are created and managed by the library. Each kernel-level thread issues a read() or write() system call for a request. Because read() and write() calls are synchronous, the thread sleeps until the I/O operation completes. Note that the sleeping thread does not interfere with other threads. The thread is awakened by the interrupt handler when the I/O is done. Obviously, the number of kernel threads in the library depends on how many asynchronous I/O requests are outstanding at a given time, and the number needs to be changed dynamically.

One may think that applications can directly use kernel-level threads to avoid blocking on I/O calls. Although this is possible, it has a disadvantage over the library approach because each application has to handle the kernel-level threads explicitly, which is an extra overhead of writing the application. The library approach does not incur the extra overhead. With the library approach, applications can simply issue asynchronous I/O calls without knowing about kernel-level threads.

The library approach has several advantages:

- No change is required in device drivers or other parts of the kernel to handle asynchronous I/O.
- The library can support any type of file.
- The implementation is relatively easy because it is outside the kernel.

However, a major disadvantage is that the library fails to preserve the order of requests passed to it. This is because kernel threads are scheduled separately and the scheduler has no information on how the threads need to be ordered. Some devices such as tapes need to read and write data in the order of the requests. If the order is not preserved, the tape driver is not able to process the data correctly. Ordering also affects the efficiency of sequential reads and writes. If sequential requests are put into the device driver's queue in order, the elevator algorithm in

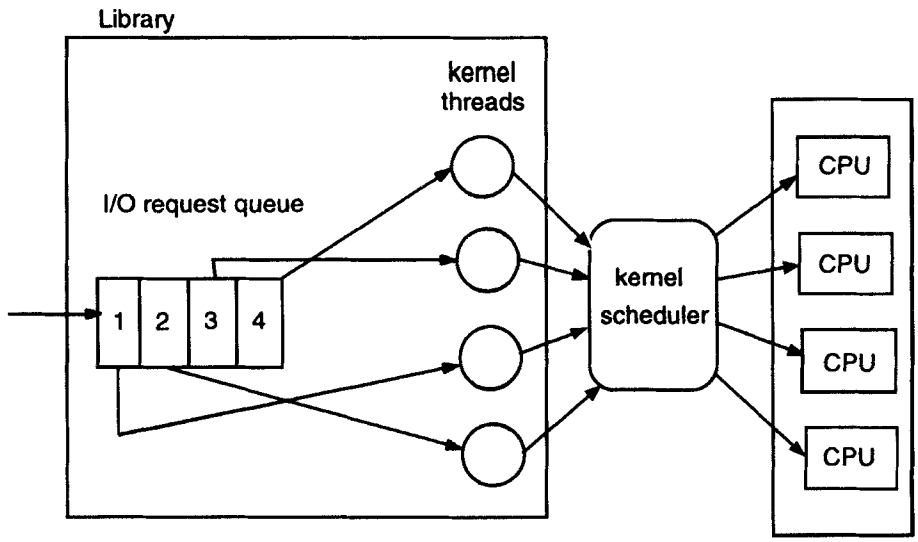

Figure 1. Library approach

the driver may process the requests in one swing. If they are not in order, the disk arms would have to swing many times.

Another disadvantage is that there is potential performance degradation due to the overhead of thread manipulation and the switching between threads in the library. The premise of the library approach is that the thread switching incurs low overhead. This thread switching overhead is evaluated in the experiments section. The library approach is also limited only to those systems that support kernel-level threads. This is another disadvantage of the library approach.

KERNEL APPROACH

The basic concept of kernel-based asynchronous I/O is not to maintain queues in user address space but to put I/O requests directly into device driver queues. When asynchronous read or write requests come in, the kernel needs to do the same things that it does for synchronous I/O read or write call, except it should not call biowait() after passing the request to a proper device driver routine.

An issue in the kernel approach is how the kernel associates requests with the proper callers without waiting for the completion of the requests. A simple way of achieving this is that before invoking the driver routines the kernel records the caller's identity and stores the information of a request, such as size and offset. For this book-keeping, the kernel maintains a list of internal buffers, which we call asynchronous I/O slots (AIO slots). The AIO slots are allocated one per request as follows. When an asynchronous read or write call is made, the kernel first checks the permission of the file descriptor, then allocates an AIO slot and saves the caller information in an AIO slot. Then, the strategy routine is invoked with the pointer of the allocated AIO slot. Upon completion of the request, the interrupt handler puts the AIO slot into a completion queue, and sends notification such as signal.

In the implementation of the kernel approach, there are three alternatives for the interaction between the file system layer and device drivers. The first alternative is that the file system layer invokes the proper driver strategy routine directly, without going through device driver read and write entry points, because invoking the entry points causes the caller to sleep. The second is to overload the existing read and write entry points. The file system layer will pass a new flag to read and write entry points in order to indicate that the request being passed is an asynchronous I/O request. The third is to define new driver entry points for asynchronous I/O.

In the beginning, we were attracted by the first alternative because the implementation could be done without changing any of the device drivers, so that drivers do not have to know about asynchronous I/O. However, we were concerned that the alternative bypasses the DDI/DKI framework so that it could break some of the existing device drivers. The second alternative looked fine, but we did not feel easy about overloading existing interfaces because this tends to be error-prone.

Finally, we chose the third alternative — adding new entry points for asynchronous I/O called aread and awrite. The main motivation was that it ensures the compatibility with the existing device drivers. However, an implication is that device drivers have to define the new entry points in order to take advantage of asynchronous I/O. In other words, unless an existing device driver is changed to export the new entry points, the driver will not be able to use asynchronous I/O. Figure 2 describes the

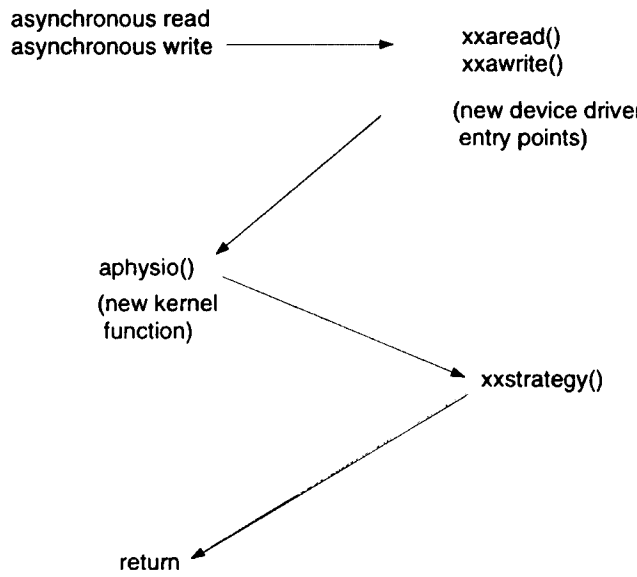

Figure 2. Kernel approach

new entry points and their interaction with the kernel. The new kernel function aphysio() calls the driver strategy routine and returns without calling biowait().

A disadvantage of the kernel approach is that the kernel has to maintain AIO slots, which takes up kernel memory. We allocated the AIO slots dynamically because we did not want to reserve kernel memory just for the slots. Therefore, this may lead to the possibility that using asynchronous I/O would exhaust kernel memory in the system. To avoid this possibility, we recycled the AIO slots after they were freed. We also added a throttling mechanism to keep a certain amount of free memory in the system.

An advantage of the kernel approach is that I/O requests are processed in the order that they are issued until the I/O requests are handed to the device driver. This is better than the library approach where the order is not preserved once kernel threads enter the kernel. However, the kernel approach does not guarantee that the order of requests will be preserved to the level of the device itself, because device drivers may rearrange the requests in their queues. Another advantage of the kernel approach is efficiency: it does not involve any context switching induced by I/O requests because the caller does not block after issuing an I/O request.

EXPERIMENTS

The experiment environment is the Solaris 2 operating system which is Sun's SVR4 implementation. It includes a library called libaio that implements the library approach described earlier. We found that the library spawns up to 50 kernel-level threads as needed. We implemented the kernel approach explained in the previous section. The objective of the experiments was to measure the performance characteristics of the library approach (libaio) and kernel approach (kaio). Rather than measuring the timing of asynchronous reads and writes in libaio and kaio, we wanted to measure

real throughput to show the performance differences. So we chose two commercial applications which heavily exercise asynchronous I/O but use it differently. One application (application A) is user-level multithreaded so that asynchronous I/O is the only choice for I/O operations. The application itself does thread switching and thread scheduling. The other application (application B) uses an I/O process which issues asynchronous I/O requests in order to enhance the throughput. We believe that these two are representative applications of asynchronous I/O in the real world.

Running a copy of an application would hardly reflect realistic utilization of asynchronous I/O. So we designd the experiments by varying the numbers of clients who request asynchronous I/O, and by varying the number of processors on which the applications run. The performance metric involves the number of iterations per second of a loop that consists of reading, modifying, and writing back data on disks.

Figures 3 and 4 show the performance of application A over various numbers of clients with libaio and kaio. Due to the internal problems of the application, we were able to run application A only on single and dual processor machines. The square points in the figures are for the measurements of kaio, and the circles are for libaio. In Figure 3, the performance of both libaio and kaio increases fairly steeply and reaches peak performance at 30 clients, and gradually decreases beyond 30 clients. But kaio performs better than libaio for the whole range of clients in the experiments. The improvement of the peak performance by kaio over libaio is 10 per cent. Figure 4 shows the results on a dual processor machine. The overall shapes of the graphs are similar to Figure 3. The peak performance of kaio is higher than that of libaio by 12 per cent. The ratio of the measured number of context switching between libaio and kaio is 4 : 1.

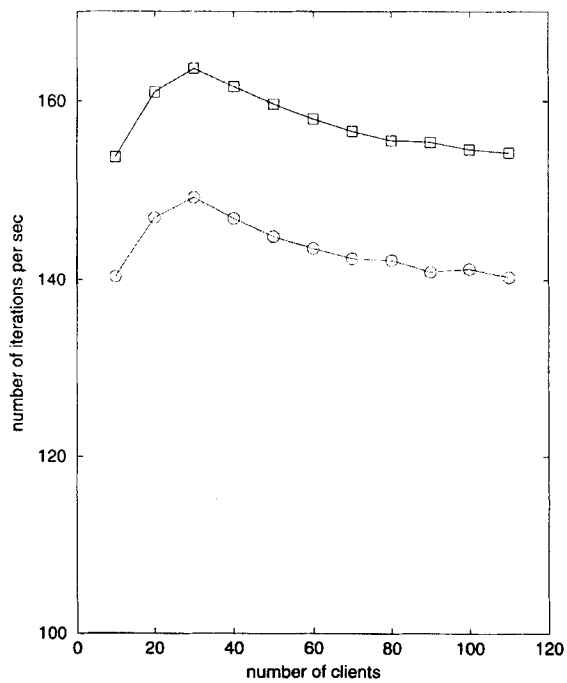

Figure 3. Varying the number of clients of application A with single processor

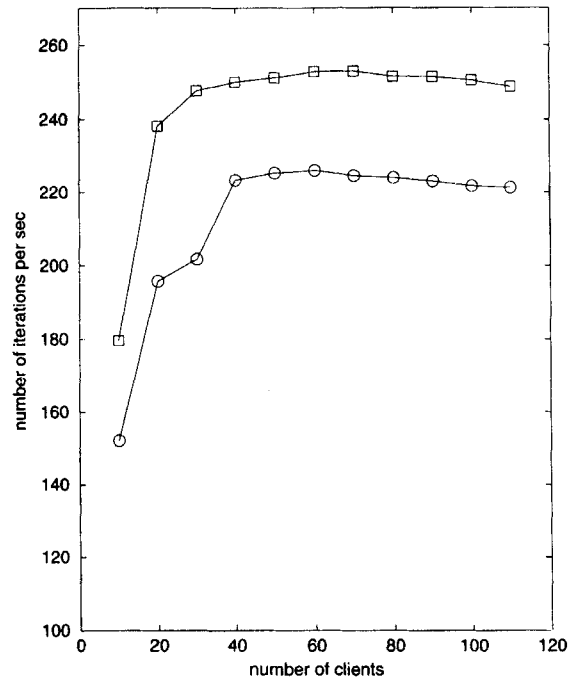

Figure 4. Varying the number of clients of application A with dual processors

Figures 5 and 6 are the results of application B. The performance results for varying the number of processors from two to 20 are shown in Figure 5. For each data point of Figure 5, we ran a series of experiments with a different number of clients to find the peak performance for the given number of processors. The results show an improvement of kaio over libaio from 6 per cent (on two processors) to 30 per cent (on 14 processors). It is interesting that the libaio performance reaches its peak at 16 processors and decreases after that, whereas the performance with kaio increases almost linearly up to 20 processors. Figure 6 shows how the performance changes over the number of clients on a 20 processors machine. The results show up to 21 per cent better peak performance via kaio as opposed to libaio.

CONCLUSION

We compared two approaches to accomplishing asynchronous I/O in UNIX. First, we investigated the differences in the design and implementation of the library and the kernel approaches. Then, we ran two real applications on the Solaris 2 operating system, with the library and kernel implementations, and measured the performance difference by varying the number of clients and processors. Although the library approach has advantages of being general and being implemented outside the kernel, we found that it incurs more overhead than the kernel approach. The results of experiments show that the kernel approach gives better performance than the library approach over the wide range of clients and processors. The peak performance difference is between 10 and 30 per cent, and the difference has been observed consistently on both applications.

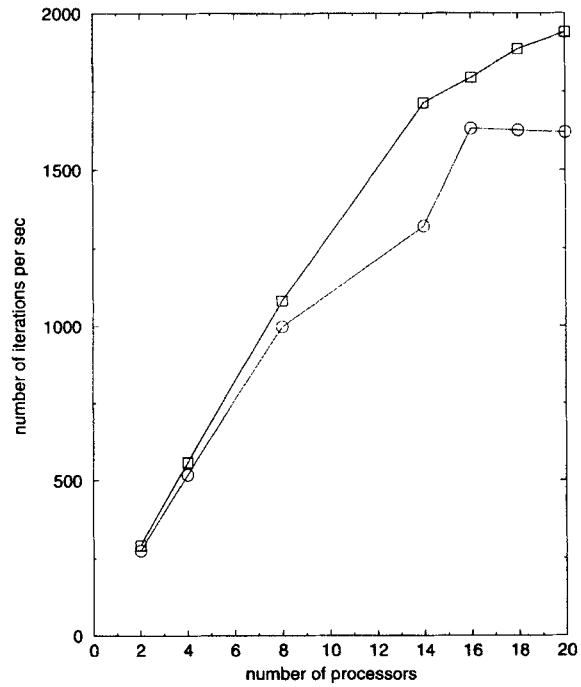

Figure 5. Varying the number of processors with application B

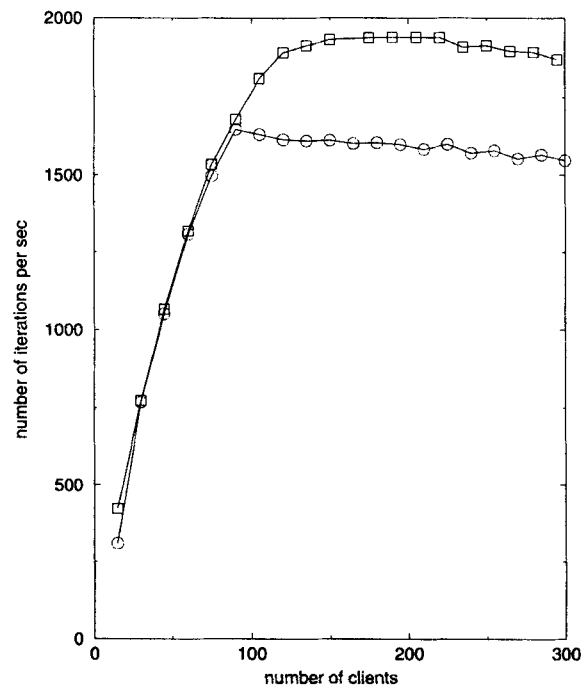

Figure 6. Varying the number of clients of application B

Although the experiments have been done on a specific environment, we believe that the performance difference is generic because it is mainly due to the overhead like context switching, and such overhead is hard to avoid in the library approach.

ACKNOWLEDGEMENTS

We would like to thank the DDI group in Sunsoft, including Jerry Gilliam and John Lee for discussing DDI issues. Dan Stein is the author of libaio in Solaris. He and Joe Eykholt provided thoughtful feedbacks to the design and implementation of the kernel approach. Greg Slaughter at Sun and Professor Insup Lee at the University of Pennsylvania provided helpful suggestions on an early draft. Special thanks to Carlos Godinez and Keng-Tai Ko, who set up and ran the experiments. We are grateful to the anonymous reviewers who provided many useful comments that improved the paper.

REFERENCES

1. H. Yoo and T. Rogers, UNIX Kernel Support for OLTP Performance. In *Proc. 1993 Winter USENIX Conference*, pp. 241–247, 1993.
2. M. Bach, *The Design of the UNIX Operating System*. Prentice-Hall, 1986.
3. UNIX System Laboratories, *System V Release 4 Device Driver Interface/Driver Kernel Interface Reference Manual*, 1992.
4. IEEE Computer Society, *POSIX Realtime Extension, P1003.4*, 1994.
5. Digital, *VAX/VMS: Internals and Data Structures*.
6. Pyramid Technology Corp., *OXs 4BSD Manual Pages. Vol. II*. 1990.
7. A. Buck and R. Coyne, An Experimental Implementation of Draft POSIX Asynchronous I/O. In *Proc. 1991 Winter USENIX Conference*, pp. 289–306, 1991.
8. M. Accetta *et al.*, Mach: a new kernel foundation for UNIX development. In *Proc. 1986 Summer USENIX Conference*, 1986.
9. D. Stein and D. Shah, Implementing Lightweight Threads. In *Proc. 1992 Summer USENIX Conference*, 1992.
10. M. Powell *et al.* SunOS Multithread Architecture. In *Proc. 1991 Winter USENIX Conference*, 1991.