

Block Asynchronous I/O: A Flexible Infrastructure For User-Level Filesystems

Muthian Sivathanu, Venkateshwaran Venkataramani,
and Remzi H. Arpaci-Dusseau

University of Wisconsin-Madison

Abstract *Block Asynchronous I/O (BAIO) is a mechanism that strives to eliminate the kernel abstraction of a filesystem. In-kernel filesystems serve all applications with a generic set of policies, do not take advantage of application-level knowledge, and consequently deliver sub-optimal performance to a majority of applications. BAIO is a low-level disk access mechanism that solves this problem by exporting the filesystem component of the kernel to the application level, thereby facilitating construction of customized user-level filesystems. The role of the kernel is restricted to regulating access to disk by multiple processes, keeping track of ownership information, and enforcing protection boundaries. All other policies, including physical layout of data on disk and the caching and prefetching of data, are implemented at application-level, in a manner that best suits the specific requirements of the application.*

1 Introduction

Generality and performance have long been at odds in operating systems [1]. Often designed as general-purpose facilities, operating systems must support a wide class of potential applications, and the result is that they offer reasonable performance for a majority of programs, while providing truly excellent performance for only a very few.

Recent work in application-specific operating systems such as the Spin [3] and Vino [8] has addressed this inherent limitation by restructuring the operating system to be *extensible*. Applications running on top of such systems can tailor policies of the OS so as to make better-informed decisions, and thus can improve performance by orders of magnitude in certain scenarios. However, these efforts all mandate that an entirely new operating system structure be accepted and widely deployed in order for such functionality to become available, which may not be realistic [2].

Of particular importance to a large class of important applications, including web servers, file servers, and databases, is the performance of the file system. To achieve optimal disk I/O performance, it is imperative that applications have full control over all aspects of the layout and management of data on disk. Applications often have a complete understanding of the semantics of the data they manage, and hence are the ideal candidates to dictate decisions on how the data is to be laid out on disk, what data is to be cached, and when prefetching should occur.

In this paper, we describe Block Asynchronous I/O (BAIO), an in-kernel substrate that enables direct access to disk. User-level library-based file systems can be built on top of BAIO, and thus can deliver high performance to applications by exploiting application-specific knowledge. The role of the kernel is limited to regulating access to the disk by multiple processes and imposing protection boundaries. BAIO differs from the large body of work on extensible operating systems in that it only requires small modifications to a stock Linux kernel.

There are three keys to the implementation of BAIO: an asynchronous interface, limited but powerful control over disk placement, and a flexible caching scheme. Asynchrony enables the construction of highly efficient I/O-bound applications without the overhead of threads, can be used to overlap I/O latency with other operations, and can increase the effectiveness of disk scheduling. Control over layout allows applications to place related data items near one another, and thus minimize disk seek and rotational overheads. Flexible caching allows applications to place data likely to be accessed by many applications or across many runs in a shared global cache, and to turn off caching (or cache in the application itself) data that is likely important only to a particular run of the application. In combination, these three features combine to allow applications or user-level filesystems built on top of BAIO to extract a high level of performance from the underlying disk system, by specializing their file layout and access to best suit their needs.

The rest of this paper is organized as follows. In Section 2, we discuss conventional disk I/O methods and their disadvantages. In Section 3, we introduce BAIO and explain its advantages as compared to conventional methods, and in Section 4, we discuss implementation details. Finally, in Section 5, we provide a performance evaluation, discuss related work in Section 6, and conclude in Section 7.

2 Conventional I/O

There are predominantly two ways in which disk I/O is conventionally performed. One is through an in-kernel filesystem, and the other is through the raw-disk interface provided by most UNIX systems. Both these methods suffer from major limitations. As discussed above, filesystems suffer due to their generality, as they are designed to be “reasonable” for a majority of applications, rather than being “optimal” for a specific one. For example, filesystems commonly have a read-ahead scheme where they prefetch a certain number of blocks that are contiguous to the requested block. This policy fails to consider the fact that the access patterns of applications need not necessarily be sequential, and in such a case, the additional prefetch that the filesystem does is unnecessary work that only degrades performance. I/O-intensive applications such as web servers and databases suffer greatly from this generality, as these applications have highly-specialized requirements [9].

To accommodate specialized applications like those mentioned above, many UNIX systems provide a raw-disk interface, allowing applications to gain direct control over a logical disk partition. Though this eliminates many of the

problems due the generality of a filesystem, it has its own set of limitations. First, access to raw disk is limited to privileged applications, so non-privileged applications must still use the filesystem. Second, the granularity of access is a whole device partition, and hence multiple applications cannot co-exist within the same partition. Third, there is no notion of ownership within a partition.

Thus filesystems and the raw disk are two extremes, one in which the kernel provides a full I/O service, and the other in which it does not even provide the basic OS service of multiplexing. A mechanism that leaves policy decisions to the user level but still manages ownership and protection boundaries across applications would be clearly more effective and flexible than either of these schemes.

A limitation that both the filesystem interface and the raw-disk interface have in common is the fact that these interfaces are essentially synchronous, *i.e.*, an application that initiates an I/O request is blocked until completion of the I/O. Considering that disks perform several orders of magnitude slower than the CPU, a blocking I/O mechanism incurs a great performance penalty. Though the impact of a synchronous mechanism may not be drastic in a heavily multiprogrammed workload where the overall throughput of the system is more important rather than that of individual applications, the impact is quite significant in dedicated systems like web-servers and database systems. For example, it is not reasonable to impose that a web-server application be blocked on I/O to fetch a page from its disk, thereby disabling the server from accepting or servicing subsequent connections in the meantime. What would clearly be more desirable, is a mechanism where the disk I/O and processing could be overlapped. Currently, this is being done by employing multiple threads to service different I/O requests, but this model of multiple threads scales very poorly and the overhead of context-switching and management of threads very soon becomes a performance bottleneck. A better solution would be an interface that exposes the inherent asynchrony of disks to applications. With such an interface, the process issuing an I/O request is returned control immediately and is notified by the kernel on completion of the I/O. An asynchronous interface will also facilitate application-specific prefetching of data from disk.

3 Block Asynchronous I/O

3.1 Overview

Block Asynchronous I/O (BAIO) is a solution aimed at addressing the above-mentioned limitations in performing disk I/O through conventional methods. BAIO is an in-kernel infrastructure that provides a direct, protected interface to the disk. With this mechanism, applications can specify which disk blocks to read or write, and the kernel only takes care of assigning capabilities to disk blocks and ensuring protected access by the applications. All other policies with regard to organization and management of data on disk are left to the user applications. In effect, applications that utilize BAIO have the power and flexibility to create their own filesystems customized to their peculiarities and specific requirements.

With BAIIO, applications also have precise control over how data is laid out on disk. Since the BAIIO interface is at the disk-block level, applications can place closely related data in physically contiguous disk locations, thereby ensuring that access to such data incurs minimal seek and rotational overhead.

The interface that BAIIO provides is asynchronous, which makes it flexible and efficient, because applications can overlap disk I/O with useful processing without incurring the additional overhead of multiple threads of control. Moreover, this enables applications to implement customized prefetching policies. Applications can decide, based on the semantic knowledge available to them, which data blocks are accessed together. This notion of “semantic contiguity” of data is more accurate than the “physical contiguity” that generic filesystems try to exploit. Because applications know the exact location of their data, they can prefetch those blocks that contain semantically contiguous data, which have a very strong likelihood of being accessed given knowledge of application access patterns. In doing so, the performance degradation due to the *ad hoc* prefetching techniques of the kernel is avoided. Note that the benefit of customized prefetching is difficult to realize even with the raw-disk interface, because of its synchronous nature.

Applications can also decide the extent of caching required and the exact disk blocks to be cached. In-kernel filesystems utilize a global buffer cache and treat all data blocks equally (likely caching those that have been accessed recently), whether or not those blocks are likely to be accessed again. With BAIIO, applications or user-level libraries can decide which blocks to cache based on specific knowledge of access patterns. The benefits of application controlled caching have already been studied [2], and an exhaustive discussion on the various benefits is beyond the scope of this paper. By providing for application-specific caching, BAIIO allows applications to take advantage of these benefits.

Though BAIIO provides a direct interface to the disk to applications, multiple applications can co-exist and share a single logical device, as opposed to the raw-disk interface where the unit of ownership is a whole device. The kernel keeps track of fine-grained ownership and capability information for different sets of disk blocks and ensures that protection boundaries are not violated – applications can only access those blocks that they are authorized to access.

3.2 Basic Operation

An important aspect of the BAIIO interface is the notion of a “disk segment.” We define a disk segment as a sequence of physically contiguous disk blocks. The application sees a disk segment as a logically separate disk with a starting block number of zero. The logical block numbers are not the same as the actual physical disk block numbers, but blocks that are contiguous with respect to logical block numbers are guaranteed to be physically contiguous. We arrived at this interface because we felt that applications are more concerned with ensuring that data accessed together are physically contiguous on disk and do not generally need to decide or know about the exact physical position in which data is stored. Thus, regardless of where a disk segment is mapped onto the physical disk, blocks in a disk segment are guaranteed to be physically contiguous, and therefore any

decision that the application makes based on logical contiguity of disk blocks will remain valid in the physical layout also.

An application makes a request to the kernel for a disk segment of a specified size, upon which the kernel looks for a physically contiguous set of free disk blocks of the desired size, and subject to quota and other limitations, creates a capability for the relevant user to the disk segment. Subsequently, the application can open the disk segment for use, whereupon it is granted the capability to that disk segment by the kernel. This capability is valid for the lifetime of the application, and the capability allows the application to perform I/O on the disk segment directly by specifying the exact blocks to read or write. Each read or write request must also include the capability that is granted during open. A capability is akin to a file descriptor in UNIX for normal files. As the BAIO read/write interface is non-blocking, the process regains control immediately after issuing a request, and is notified upon completion of the I/O.

Multiple per-block read and write calls can be merged into a single BAIO operation, which has two primary advantages. First, the disk driver can create a better schedule of disk requests, minimizing seeks and rotational overhead. The more requests the driver has, the better job of scheduling it is likely to do (note that the asynchronous interface BAIO provides also helps in this regard). Second, system call overhead is reduced, as merging reduces the effective number of system calls an application invokes. Thus, the application can control the number of user-kernel crossings, in a manner similar to but more flexible than the `readv()` and `writev()` system calls.

By giving applications total control over a disk segment, BAIO creates the potential for every disk segment to have its own filesystem, tailor-made to the specialized requirements of the application that manages it. Traditional filesystems can be implemented on top of BAIO for the large class of applications that do not require the sophistication of an interface like BAIO. Such applications can simply link with the standard filesystem implementation and operate as before. Another possible approach to support backwards compatibility is to deploy file-server applications each implementing one standard filesystem. Applications can connect to whichever file server they are interested in, and read and write calls can be transparently redirected through stubs to the appropriate file servers. This will permit existing applications to be easily adapted to run on top of BAIO, since it will only involve relinking of the object binary with the appropriate user-level filesystem implementation.

3.3 Structure of a BAIO Device

In the kernel, BAIO keeps track of information pertaining to various disk segments, including the mapping of a disk segment to the physical disk, ownership attributes for disk segments, and so forth. This information is held in a per-disk-segment structure called an inode. The inode also contains the name of the disk segment within itself. This is in contrast to the UNIX style of having names and their inode numbers in directories, and was necessitated because we chose to have a flat namespace for naming disk segments, in-order to provide flexibility to the user-level filesystem to implement its own naming scheme, rather

than impose a fixed hierarchical naming scheme. Inodes are hashed by the disk segment name. The kernel also maintains global information on the free blocks available for future allocation.

The first block in a BAIO device contains the superblock structure which maintains information on the total number of blocks in the device, the block-size of the device, the number of inodes on disk and pointers to the start of free list blocks, inode blocks and data blocks. Free lists are maintained in the form of bitmaps, with a free list hint indicating which position in the free list to start looking for a contiguous stream of free blocks. Considering that disk segments are meant to be reasonably large (spanning several megabytes), this organization of free block information is reasonable, especially in light of the fact that creation of new disk segments is a rare event compared to regular open, read, or write operations.

In most cases, the free-block hint maintained enables fast location of a free chunk. Again this decision on maintenance of free block information is not as crucial in our model as it is in normal filesystems because, in the latter, free blocks have to be found every time a write is made, and therefore sub-optimal tracking of free block information will slow down all writes. In BAIO, the free list is accessed only at the time of disk segment creation and once a segment is created; applications just read or write into pre-allocated disk blocks and hence excessive optimization in this area was not warranted. The number of inodes has been fixed at 1/1000th of the total number of data blocks in the system. Clearly, the average length of segments should be large enough to amortize this fixed overhead.

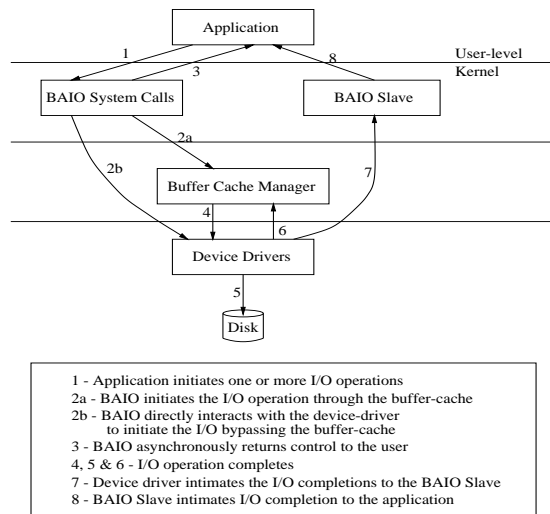


Figure 1. Control flow during a BAIO operation.

3.4 Asynchrony and Caching

The BAIO architecture achieves asynchrony by queueing I/O requests at the device and returning control to the application immediately. A BAIO service daemon takes care of intimating completion of I/O to the application. The service daemon enters the kernel through a specific entry-point and never returns. Its sole duty is to look through the request queue to find which I/O operations have completed and notify the processes on whose behalf the I/O was performed.

Since this daemon is also in-charge of copying data to the address space of the application in the event of a read operation, it needs to have access to the virtual address space of the application. There are two methods in which this could be implemented. The first method is to use in-kernel shared-memory objects to share a memory region between the application and the service daemon. In this model, there will be a single BAIO service daemon for the entire system, managing the I/O requests of all processes. The drawback of this model is that considerable overhead is incurred at the service daemon in binding to shared-memory objects and the single service daemon can become a performance bottleneck. Moreover, this would require that the application allocate a shared-memory object and attach its data buffer to the shared-memory region prior to invoking the BAIO system call. Since there is a system-wide limit on the number of shared memory objects that can exist, this naturally places a limitation on the number of BAIO requests that can be pending simultaneously. However, this method does have certain advantages in that the single service daemon has a complete view of all pending requests and hence can potentially make better scheduling decisions.

The other method, which we have chosen for our model, is to have the service daemon share the application's virtual address space. This means that each process has its own BAIO service thread and this thread can directly write into the application's address space. This is more scalable than the first alternative because now the service thread is only in-charge of BAIO requests initiated by a single process and the overhead of attaching to shared memory objects on every I/O completion is eliminated.

The overall control flow during an I/O operation through BAIO is depicted in Figure 1. There are two kinds of interfaces that BAIO provides to applications. In the first method, the I/O takes place through the global buffer cache of the kernel, which allows for sharing of caching across applications, at the cost of generality. The advantage of this method is that multiple applications will be able to share the global kernel buffer cache, which may not be possible with application-level caching alone. This method benefits a scenario in which multiple applications access the same set of disk blocks, as otherwise, each application will try to cache it separately, wasting memory. However, this method has the drawback of introducing some generality into the caching scheme for disk blocks, and could lead to double-buffering.

In the second method, I/O operations bypass the buffer cache of the kernel entirely. In this method, the application directly interacts with the device driver and does not use the buffer cache. This is similar to the raw interface provided

in UNIX systems. The option of whether or not to use the buffer cache can be specified on a per-operation basis, thereby providing for a fine-grained choice by the application. Blocks which the application thinks will be shared by other processes can be made to go through the buffer cache while others may be read/written directly. For example, if there is a large read to be made, the application may wish to ensure that this read does not flush the current contents of the buffer cache, and can therefore use the non-buffered interface.

4 BAI0 Implementation

The BAI0 model has been implemented in the Linux kernel. The interface is provided in the form of a set of system calls. Additionally, a utility for configuring a disk device for BAI0 has been implemented. A process using BAI0 keeps track of additional information in its process control block, via a block descriptor table, which is similar to the file descriptor table in UNIX. Each table entry keeps track of information such as a pointer to the inode of the disk segment, a reference count indicating number of processes that have the disk segment open, and so forth. Additionally, a process also maintains a pointer to its service daemon. The service daemon maintains a queue of BAI0 requests issued by its master process and are pending. Processes also maintain synchronization information for regulating access to the request queue which is shared between the master process and the service daemon.

When an application invokes the BAI0 system call, the process adds the request to the request queue of its service daemon and issues the request to the buffer cache layer or the device driver, as was required by the application. It then returns control to the application immediately. The disk driver, on completion of the I/O, wakes up the service daemon. On waking up, the service daemon looks through its request queue to find out which of the requests have completed, and accordingly notifies the application. Notification can be done in two ways. The first is to set a status variable in the application address space. The second is to signal the application on completion of I/O. A signal-based scheme would substantially complicate the application code since it must be prepared to process completion of I/O requests in signal handlers. Moreover, signals may be lost if the application spends too much time in handling a request. The first scheme is simpler and more scalable than a signal based scheme, and therefore, has been chosen in our implementation.

The set of system calls which constitute the BAI0 interface to the application level are:

1. **create_dseg**: takes a device name, segment name and a segment size as input, allocates a disk segment of the required size if possible
2. **open_dseg**: opens an existing disk segment specified by the name and device, and returns a descriptor to the segment
3. **baio**: this is the call used to perform I/O. Multiple read/write requests can be specified as part of a single BAI0 call; includes a block descriptor as input

4. `baio_service`: this is the entry-point to the service daemon. The service daemon invokes this syscall and never returns.
5. `baio_mount`: this call takes a device name as argument, and if the device is configured for BAIO, mounts the device.

5 Performance Evaluation

The performance enhancement possible by allowing application control over various aspects of filesystem policies has been studied in great detail. For example, Cao *et al.* show that application-level control over file-caching can reduce execution time by around 45% [5]. In another study, Cao *et al.* measured that application-controlled file caching and prefetching reduced the running time by 3% to 49% for single-process workloads and by 5% to 76% for multiprocess workloads [4]. By exporting the entire filesystem policy to the user-level, we enable applications to obtain all these performance advantages.

The measurements we have taken mainly evaluate the impact of exposing the asynchrony of disks to applications. We compare the performance of our model with the UNIX raw disk implementation and another asynchronous I/O technique developed by SGI called Kernel Asynchronous I/O (KAIO). Experiments were performed with and without the effect of buffer cache. In the first set of experiments, we compared non-buffered BAIO, raw disk, and KAIO over raw disk. In the next, we compared buffered BAIO, ext2 and KAIO over ext2.

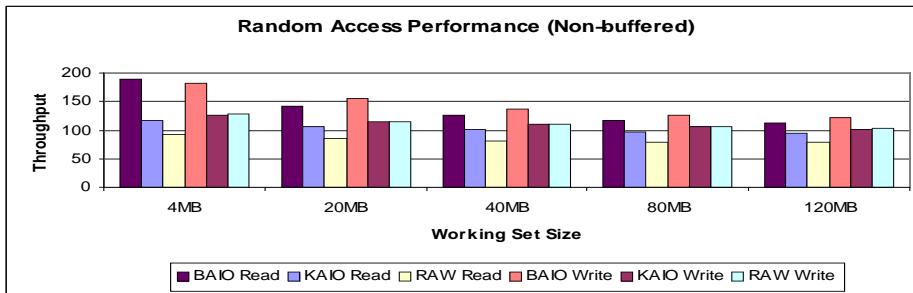


Figure 2. Non-buffered random reads and writes.

Considering that an asynchronous interface will benefit those applications that can overlap disk I/O with useful processing, we have chosen a workload that resembles that in a web-server. There is a continuous stream of requests, and each request will involve a disk I/O and some processing on the data read/written. In the asynchronous interfaces, we pipeline the disk I/O and processing over several stages. Stage i will process the data corresponding to the $(i-k)$ th request while issuing the I/O for the i th request. This way, by the time data is required for processing, it will already be available since I/O for that data was initiated quite in advance. In other words, the application does not spend any time waiting for I/O to complete. In our studies, as expected, we find that both BAIO and

KAIO consistently outperform raw I/O and ext2. BAIO performs much better than KAIO as well. The reason for this is that while BAIO exposes the full asynchrony of the disk to applications, KAIO employs slave threads which invoke the blocking I/O routines. Hence the asynchrony of KAIO is limited to the number of slave threads in operation. Having a large number of threads is not a solution because, as the number of threads increases, system performance degrades due to the additional overhead of context switches and thread management.

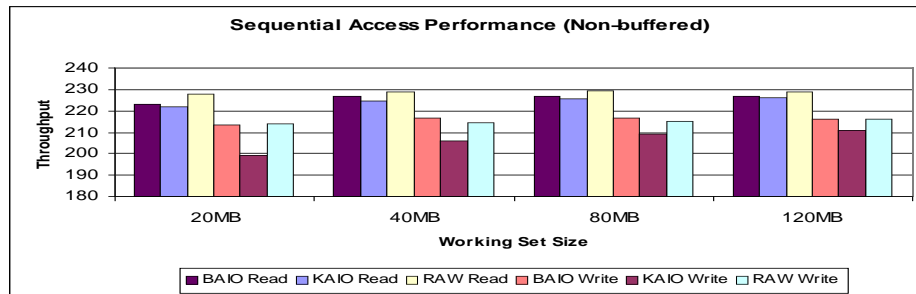


Figure 3. Non-buffered sequential reads and writes.

The performance metric we have chosen in all the three graphs presented, is the average throughput of the application, i.e the number of requests serviced by the application in one second. Figures 2 and 3 show the behavior of random and sequential reads and writes without the effect of buffer cache (each read/write involved a 8K data transfer, from/to disk blocks randomly chosen within the working set size). In this case, BAIO with buffer cache disabled was compared with KAIO running over raw disk, and plain raw I/O. We observe that in the random access case, BAIO performs on an average 25% better than KAIO and 54% better than raw I/O. This is possible because BAIO allows the disk latency to be partially hidden by enabling applications to overlap disk I/O with processing, so that by the time the results of an I/O are required, they are most likely to be already available, thereby eliminating the time the application spends in waiting for I/O. In sequential access, BAIO outperforms KAIO, but raw I/O performs marginally better than BAIO. The reason we believe this happens is because sequential raw I/O is inherently fast and not much is gained by asynchrony – instead the overhead of an external process intimating I/O completion and additional context switching brings down the performance of the asynchronous schemes. This has the obvious implication that exposing asynchrony is of greater benefit for a random access workload than one of sequential access.

In the next study, shown in Figure 4, we compared BAIO with all requests going through the buffer cache, KAIO over ext2 and plain ext2. We observe that BAIO consistently outperforms KAIO and raw I/O in random reads, but in random writes, ext2 and KAIO perform better. This is because of the delayed write strategy of ext2, which also applies to KAIO since the slave threads of KAIO internally use ext2 I/O calls. Since writes are fully buffered and only flushed to

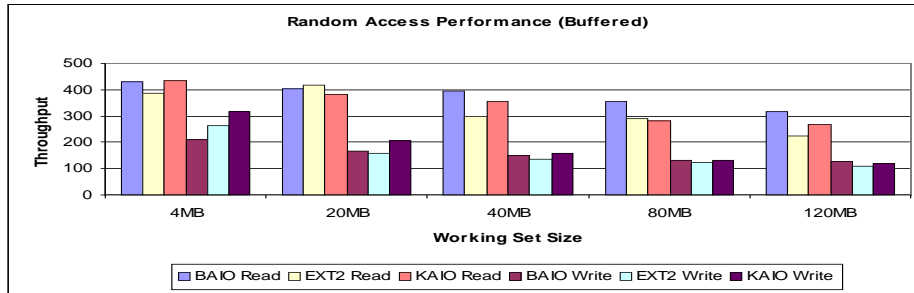


Figure 4. Buffered random reads and writes.

disk once at the completion of the experiment, many writes to the same block are absorbed by the buffer cache. Moreover, the sync is highly optimized because of the large number of blocks to write to disk, thereby facilitating optimal disk scheduling. In spite of all these, for very large working sets, exceeding 80 MB, BAIO outperforms ext2 and KAIO, since at this size, the buffer cache tends to get filled up and more writes go to disk. Thus, in all these studies, BAIO clearly achieves better performance than the two other supposedly efficient and powerful interfaces.

6 Related Work

The idea of minimizing or eliminating operating system abstractions is as old as the concept of micro-kernels are. There have been numerous contributions that argue for a minimal operating system with policies exposed to the applications. Nucleus [7] and Exokernel [6] are two classic examples of such models. BAIO draws inspiration from these contributions and applies the principle in the restricted context of a filesystem. What is different in BAIO compared to prior works is that BAIO can fit into an existing operating system with very few modifications, as opposed to ideas like Exokernel which warranted a total “start from scratch” revamp. The performance advantages of exposing certain filesystem policies like caching and prefetching to the application level have been studied by Cao *et al.* [5,4]. BAIO takes this to an extreme by allowing applications to dictate virtually every aspect of filesystem policy and operation, thereby providing applications with the highest possible performance advantage achievable through customization.

The idea of providing an asynchronous interface to disk I/O is not new. The FreeBSD implementation of the AIO library is an example of an attempt to provide asynchronous disk I/O, and is similar in spirit to KAIO. Both of these interfaces achieve asynchrony by employing slave threads, which perform the same blocking I/O routines of the filesystem. Though KAIO claims to use split-phase I/O to expose the full asynchrony of disk to the application, our studies indicate no significant improvement, at least in the context of the IDE disk. Hence the asynchrony is limited to the number of threads employed, which cannot increase

indefinitely since it then becomes counter-productive due to the excessive overhead in context-switching and thread management. BAIIO, because of its careful integration into the kernel and direct interaction with the disk device driver, outperforms both these models, as was shown in the performance study (KAIO claims to outperform AIO, so performing better than KAIO is equivalent to outperforming both).

7 Conclusion

In an attempt to overcome the potential performance limitations imposed by generic in-kernel filesystems, we have proposed a mechanism known as BAIIO by which the application is given full control over the way its data is both organized and managed. Application-level semantic knowledge facilitates optimized policies and decisions that perform specifically well to the peculiarities of the application. In implementing BAIIO, we take special care to ensure that the kernel does not determine any policy with regard to the management of application data on disk, other than enforcing protection boundaries across applications and maintaining the notion of ownership of disk segments. One of the main features of the BAIIO model is that it integrates quite easily into an existing operating system, in contrast to extensible systems of the past which require that entirely new kernels be designed, developed, and deployed. A remaining challenge for BAIIO-based systems is to build and experiment with a broad range of specialized user-level filesystems, and to understand how concurrently running user-level filesystems interact with one another.

References

1. T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM TOCS*, 1992.
2. A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *SOSP 18*, October 2001.
3. B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *SOSP 15*, December 1995.
4. P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and Performance of Integrated Application-controlled File Caching, Prefetching, and Disk Scheduling. *ACM TOCS*, 14(4), November 1996.
5. P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-controlled File Caching. In *OSDI 1*, November 1994.
6. D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture For Application-level Resource Management. In *SOSP 15*, 1995.
7. P. B. Hansen. The Nucleus of a Multiprogramming System. *Communications of the ACM*, 13(4), April 1970.
8. M. I. Seltzer and C. Small. Self-Monitoring and Self-Adapting Systems. In *HotOS '97*, Chatham, MA, May 1997.
9. M. Stonebraker. Operating System Support For Database Management. *Communications of the ACM*, 24(7), July 1981.