Block Asynchronous I/O - A flexible infrastructure for user-level filesystems

Muthian Sivathanu (muthian@cs.wisc.edu)
Venkateshwaran V (veeve@cs.wisc.edu)

Abstract

Block Asynchronous I/O (BAIO) is a mechanism that strives to eliminate the kernel abstraction of a filesystem. In-kernel filesystems try to serve all applications with a uniform generic set of policies, and consequently end up achieving sub-optimal performance on a majority of applications. Application level knowledge of the peculiarities of the data that is managed on disk is totally ignored by a kernel filesystem, with the result that optimizations that could exploit specific characteristics of applications cannot be done in a generic filesystem. BAIO, presented in this paper, tries to solve this 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, caching and prefetching of data, are totally implemented at the application level, in a way that best fits the specific requirements of the application.

1. Introduction

Kernel filesystems try to be efficient across a large variety of application requirements. This excessive generality that is inherent to the concept of an in-kernel filesystem, turns out to be a great perforamance limitation in many cases. Filesystems end up ignoring the vast knowledge available at the application level that might facilitate highly optimized policies and decisions. This huge information gap that exists between filesystems and the applciations result in generic policies being imposed on all applications uniformly, leading to sub-optimal performance. This performance impact is particularly significant in highly demanding, specialized applications like databases and web servers, Stonebraker [7] argued that inappropriate filesystem implementations can have a dramatic impact on the perforamance of databases.

To achieve optimal disk I/O perforamance, it is imperative that applications have full control over all aspects of the layout and management of data on disk. Applications have the complete view of the semantics of 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. Potentially, applications require the power and capability to operate their own filesystem. The kernel must ideally provide a direct, protected access to the disk, and its role must be limited to regulating access to the disk by multiple processes and imposing protection boundaries. Block Asynchronous I/O (BAIO), presented in this paper, strives at achieving these objectives.

This paper is organized as follows . In Section 2, we discuss about the conventional methods of doing disk I/O and their disadvantages. In section 3, we introduce BAIO and explain why BAIO is a better way of doing disk I/O compared to conventional methods. Section 4 detials the design of BAIO. Section 5 deals with implementation details. In Section 6 we provide perforamance evaluations and we discuss related work 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 in UNIX systems. Both these methods suffer from major limitations. As has been already mentioned, filesystems suffer because of their generality. It is a well-accepted fact in computer systems, that generality often comes at the cost of performance, and filesystems are no exception to this. Filesystems are meant to be used by a variety of applications, and therefore try to be "reasonable" to a majority of applications, rather than being "optimal" for a specific category of applications. 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 perforamance. Thus applications are forced to incur extra work, and consequent reduced perforamance, even if they don't need them. Highly I/O intensive applications like Web servers and databases suffer greatly from this generality. These applications have highly specialized requirements that much of the work a conventional filesystem does turns out to be useless and unnecessary overhead. A mechanism by which applications are allowed to totally determine the organization and management of data on disk is bound to benefit such applications substantially.

To accommodate specialized applications like those mentioned above, UNIX provides a raw disk interface, in which applications get direct control over a logical disk partition. Though this eliminates much of the problems due the generality of a filesystem, it has its own set of limitations. First, access to rawdisk is limited to superuser applications, so non-privileged applications must still use the filesystem. Second, the granularity of access is a whole device, and hence multiple applications cannot co-exist in the same rawdisk partition. Third, there is no notion of ownership within a rawdisk partition and therefore this interface is highly restrictive. Every application using rawdisk must use a separate device, which is far from being desirable. The problem lies in the fact that the rawdisk interface totally eliminates the kernel from the application's access-path to the disk. Thus filesystems and the rawdisk are two extremes, one in which the kernel does everything and the other in which it does absolutely nothing. 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 rawdisk interface have in common, is the fact that these interfaces are essentially synchronous, i.e an application that does I/O is blocked until completion of the I/O. Considering that disks perform serveral orders of magnitude slower than the CPU, a blocking I/O mechanism incurs a great perforamance 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 webservers and database servers. 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 perforamance 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 applciation-specific prefetching of data from disk.

3. BAIO - an alternative

Block Asynchronous I/O is a solution aimed at addressing the above-mentioned limitations in performing disk I/O through conventional methods. Baio is a kernel infrastructure that provides a direct protected interface to the disk, to user applications. With this mechanism, applications can specify which exact disk blocks to read or write, and the kernel just 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 now have the power and flexibility to create their own filesystems customized to their peculiarities and specific requirements. The interface that BAIO provides is asynchronous, which makes it even more flexible and efficient since applciations can overlap disk I/O with useful processing, without incurring the additional overhead of multiple threads of control. Moreover this makes it possible for the applications to implement customized prefetching policies.

With BAIO, applications can determine the exact way in which their data is to be laid out on disk. Since the interface is at a disk block level, they can layout closely related data in physically contiguous disk blocks, thereby ensuring that access to those data incurs minimal disk seek and rotational overhead. Moreover, they can decide the extent of caching required and the exact disk blocks to be cached. This is in contrast to an in-kernel filesystem where the global buffer cache of the kernel caches all blocks that are accessed, irrespective of whether the data in those blocks have a likelihood of being accessed again. But with application-controlled caching, the semantic knowledge available to applications on whether a certain data is likely to be accessed again in the near future, can be exploited to make more effective use of the cache for optimal performance. 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 applciation-specific caching, BAIO potentially empowers the application to take advantage of these benefits.

The asynchronous interface provided by BAIO also enables applications to implement tailor-made prefetching policies. Applications can decide, based on the semantic knowledge available with them, which set of data are accessed together. This notion of "semantic contiguity " of data is more accurate than the "physical contiguity" that generic filesytems try to exploit. Since applications now know the exact disk blocks that contain their data, they can prefetch those blocks that contain semantically contiguous data which have a very strong likelihood of being accessed given the current access patterns. This relieves applications of the perforamance degradation due to unnecessary prefetching of useless blocks by the kernel on an ad-hoc basis irrespective of whether the applications actually need them. This is especially useful in I/O intensive dedicated systems like web servers and databases where the disk bandwidth needs to be fully exploited for optimal performance, and unnecessary disk traffic in the form of useless prefetching is a serious drawback. This benefit of customized prefetching cannot be realized even with the rawdisk interface, because of its synchronous nature.

Though Baio provides a direct interface to the disk to applications, multiple applications can co-exist and share a single logical device, as opposed to the rawdisk 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, i.e applications access only those blocks that they are authorized to access.

4. BAIO Interface

An important aspect of the BAIO 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 feel 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, irrespective of where a disk segment is mapped exactly into the physical disk, blocks in a disk segment are guaranteed to be physically contiguous, and therefore any decision that the application takes based on logical contiguity of disk blocks will remain valid in the physical layout also.

An application requests 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. After this, 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 subsequent to this, the application can do I/O on the disk segment directly specifying the exact blocks to read/write. Each read/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. The read/write interface provided by BAIO is non-blocking and therefore enables overlapping of I/O with useful processing. The process is returned control immediately and is notified upon completion of the I/O.

Multiple per-block read and write calls can be merged into a single baio operation. In other words, the application can request multiple disk operations through a single system call. This permits the application to do internal buffering of writes and to avoid disk reads in situations where the consistency semantics of the application permit stale data to a certain extent. It also enables the disk driver to do a better scheduling of disk requests, inorder to minimize seeks and rotational overhead. This would not be possible if every single block request is immediately issued to the device driver, which will then have a lot less choice to schedule its operations. This also helps reduce the overhead of a switch between kernel and user mode during a system call, by reducing the effective number of system calls an application invokes. The asynchrony of the interface also makes this grouping of I/O requests a feasible option as otherwise, the application will incur huge delay blocking on all I/O to complete.

By giving applications total control over a disk segment, we potentially permit every disk segment to have its own filesystem, tailor-made to the specialized requirements of the application managing it. Current filesystems like Ext2 can also be implemented on top of BAIO to benefit applications that do not require the sophistication of an interface like BAIO. Such applications can simply link with a standard filesystem implementation and operate as before. Another model possible, is to have different fileserver applications each implementing one standard filesystem, Applications can connect to whichever fileserver they are interested in. Normal read and write calls can be transparently redirected through stubs to the appropriate fileservers. 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.

5. Design of BAIO

Structure of a BAIO device

The kernel keeps track of information pertaining to various disk segments, like the mapping of a disk segment to the physical disk, ownership attributes for disk segments, etc. These information are 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. This scheme was necessitated because we chose to have a flat namespace for naming disk segments, inorder to provide flexibility to the user-level filesystem to implement its own naming scheme, rather than impose a fixed hierarchical naming. Inodes are hashed by the disk segment name. The kernel also maintains global information on the free blocks available for future allocation. The structure of a baio device is shown in Figure 1.

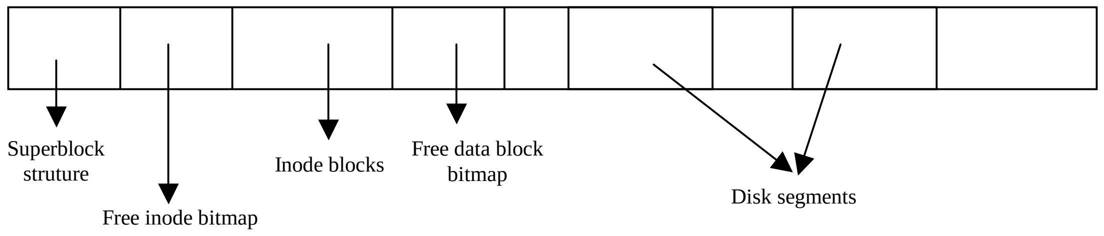


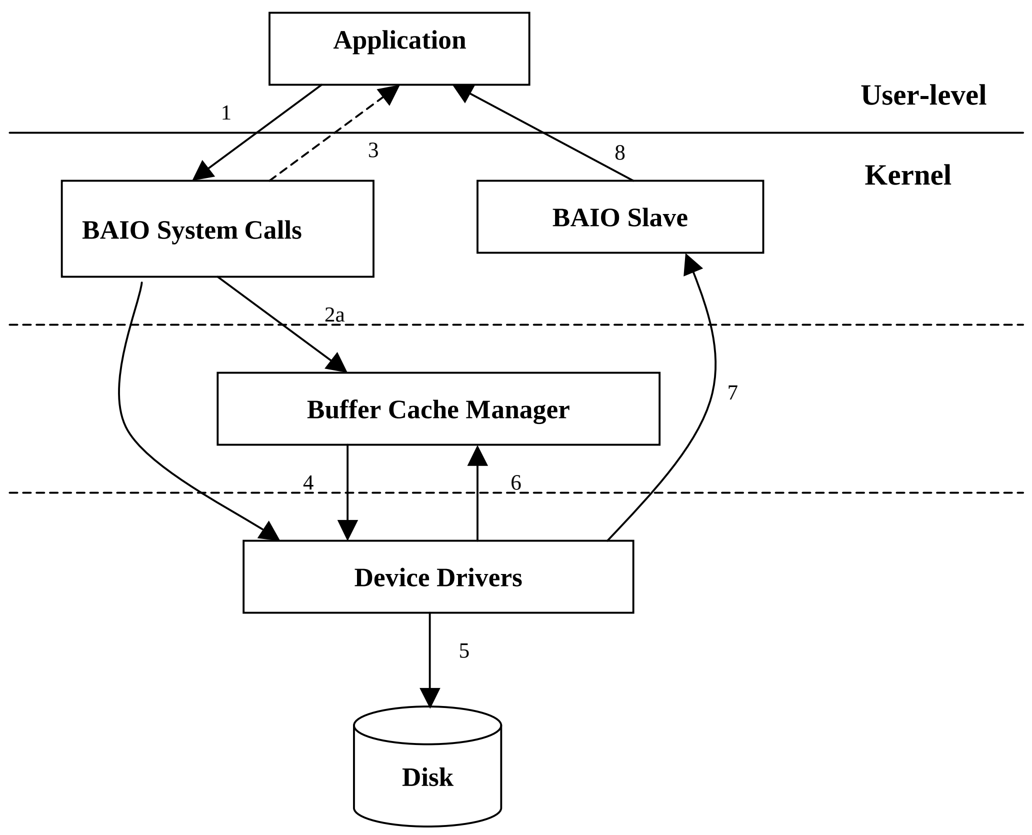
Figure 1 : Structure of a BAIO disk partition

The first block in a Baio device contains the superblock structure which maintains information on the total number of blocks in the device, the blocksize of the device, the number of inodes on disk and pointers to the start of freelist blocks, inode blocks and data blocks. Freelists are maintained in the form of bitmaps, with a freelist hint indicating which position in the freelist 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 seems reasonable, especially in view of the fact that creation of new disk segments is a rare event compared to regular open and read/write operations. In most cases, the freeblock 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 UNIX 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. But in Baio, the freelist 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 doesn't seem necessary. The number of inodes has been fixed at 1/1000th of the total number of data blocks, since we feel that the average length of disk segments should atleast be 1000 blocks, for any useful performance enhancement due to Baio. For the tailor-made filesystem policies to have any considerable effect, a disk segment should be sufficiently large to make those optimizations meaningful.

BAIO Architecture

The Baio architecture achieves asynchrony by queueing the I/O request 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. The daemon is usually in sleep mode, but gets woken up by the disk device driver when an I/O has completed. Since this daemon is also in-charge of copying data to the application's address space in the event of a read operation, it needs to have access to the application's virtual address space. There are two methods in which this could be implemented :

The first method is to use in-kernel shared memory objects of UNIX 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 very easily becomes a perforamance 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 total view of all pending requests and hence can potentially take better scheduling decisions in deciding which applications are notified first.



1 – Application initiates one or more I/O operation(s)
2a – BAIO initiates the I/O operation through the buffer cache
2b – BAIO directly interacts with the device driver bypassing the buffer cache
3 – BAIO returns control to the application asynchronously
4, 5 and 6 – I/O operation completes
7 – Device Driver intimates I/O completion to BAIO Slave
8 – BAIO Slave intimates I/O completion to application

Figure 2: Control flow during a BAIO operation

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 Fig. 2. 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. 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 an application level caching alone, where each application maintains its own local cache. 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. Our initial argument was that applciations should be able to totally determine, inter alia, the caching strategy. Using the buffer cache leads to a fixed caching and replacement strategy being imposed on applications though they use a mechanism like baio. If applications still try and implement caching internally, that would have the adverse impact of caching the same blocks twice, which leads to sub-optimal utilization of memory. This problem, called "double-buffering" is a major concern for databases.

To solve this, we also provide another interface in which the I/O bypasses the buffer cache of the kernel. In this method, the application directly interacts with the disk 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 at 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 bypassing the buffer cache. For example, if there is a huge read to be made, the application may wish to ensure that this read does not flush out the current contents of the buffer cache all at once, and can therefore use the non-buffered interface. This flexibility gives applications the much desired "total control" over how disk I/O takes place.

6. Implementation

The BAIO 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 baio has been implemented. A process using baio keeps track of additional information in its process control block, viz. a block descriptor table, which is similar to the file descriptor table in UNIX (the segment descriptor returned as a result of open_dseg is an offset into this table). Each table entry keeps track of information like a pointer to the inode of the disk segment, a reference count indicating number of processes that have the disk segment open, etc. Additionally, a process also maintains a pointer to its service daemon. The service daemon maintains a queue of baio 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 baio 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's address space. The other method 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, the possibility of a subsequent signal due to another I/O completion getting lost due to the application still executing in its handler code, makes this an unreliable scheme. 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 BAIO interface to the application level are (a) create_dseg - takes a device name, segment name and a segment size as input, allocates a disk segment of the required size if possible
(b) open_dseg - opens an existing disk segment specified by the name and device, and returns a descriptor to the segment
(c) baio - this is the call used to perform I/O. Multiple read/write requests can be specified as part of a single baio call; includes a block descriptor as input
(d) baio_service - this is the entry-point to the service daemon. The service daemon invokes this syscall and never returns
(e) baio_mount - this call takes a device name as argument, and if the device is configured for baio, mounts the device.

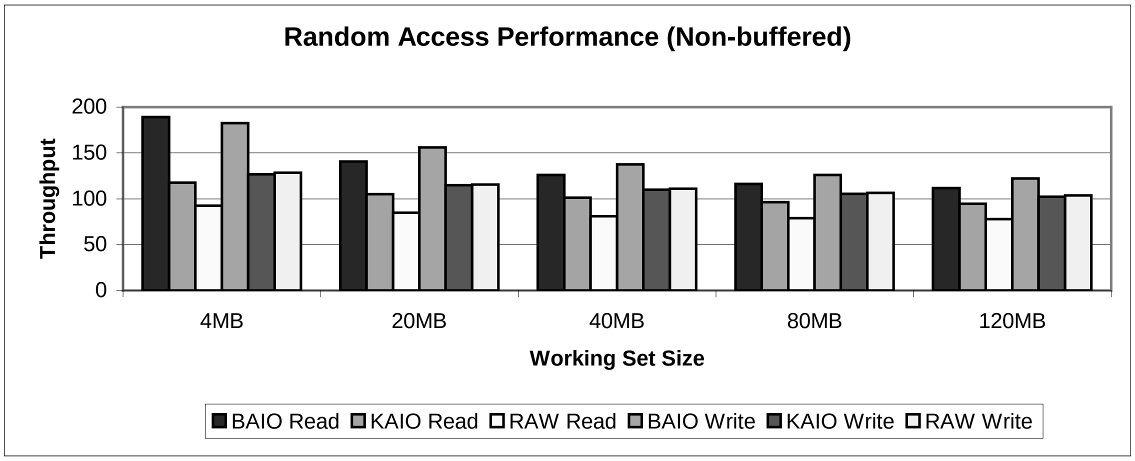


Figure 3: Behavior of non-buffered random reads and writes per second.

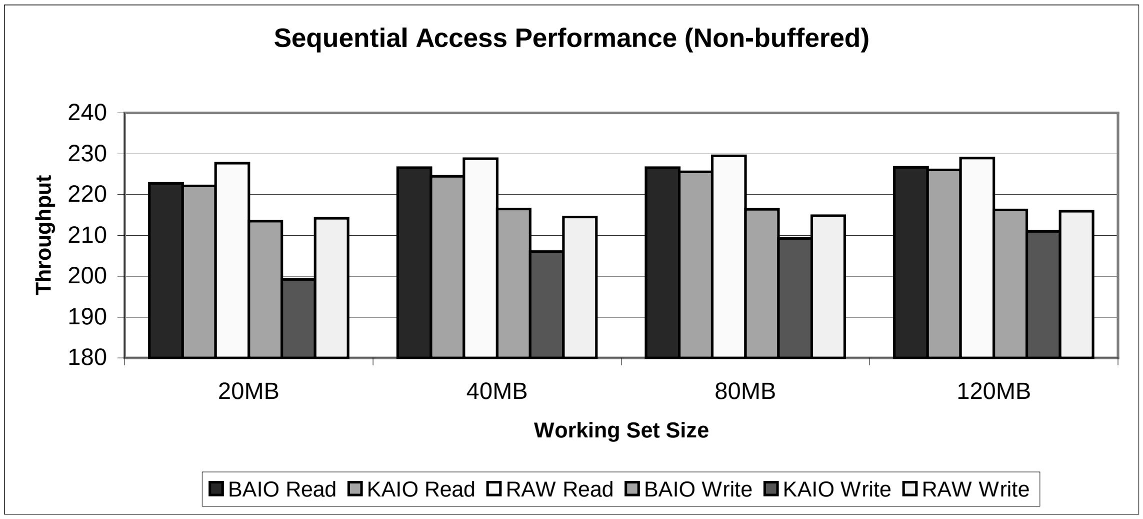


Figure 4: Behavior of non-buffered sequential reads and writes per second.

7. Performance Evaluation

The perforamance enhancement possible by allowing application control over various aspects of filesystem policies has been studied in great detail. Cao et al [2] showed that application-level control over file-caching can reduce execution time by around 45 %. In another study, Cao et al [3] 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. By exporting the entire filesystem policy to the user-level, we enable applications to incur 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, rawdisk and kaio over rawdisk. In the next, we compared buffered baio, ext2 and kaio over ext2.

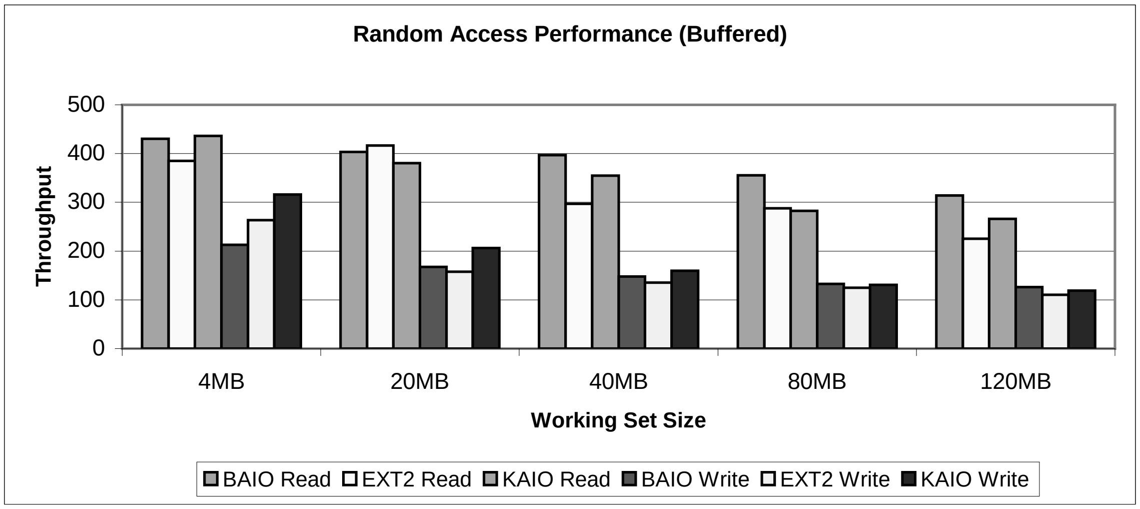


Figure 5: Behavior of buffered random reads and writes per second.

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 webserver. 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 ith 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 too. 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.

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 3 & 4 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 I/O, and plain Raw IO. 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 think this happens is because sequential raw IO 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 more beneficial for a random access workload than one of sequential access.

In the next study, shown in figure 5, 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 IO 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 a ‘sync’ is done only once during completion of the simulation, 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. Inspite 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.

Related Work

The idea of minimizing/eliminating operating system abstractions is as old as the concept of microkernels are. There have been numerous contributions that argue for a minimal operating system with policies exposed to the applications. Nucleus [4] and Exokernel [5] 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 the application level have been studied by Cao et al in [2] and [3]. BAIO takes this to an extreme by allowing applications to dictate virtually every aspect of a filesystem's 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 totally new either. FreeBSD implementation of the AIO library is an example of an attempt to provide asynchronous disk I/O. More recently, SGI has come up with an implementation of KAIO (Kernel Asynchronous I/O) [6]. Both 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 optimization atleast 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. BAIO, 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).

Conclusion

In an attempt to overcome the potential performance limitations imposed by generic in-kernel filesystems, we have proposed a mechanism by which the application is given full control over the way its data is to be organized and managed. Application-level semantic knowledge facilitates optimized policies and decisions that perform specifically well to the peculiarities of the application. A generic kernel filesystem cannot benefit from the extensive information available at the application level. We have taken 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. A nice feature of this model is that it fits very easily into an existing operating system, unlike most works in the past which often entailed phenomenal changes to the existing infrastructure. The asynchrony we have provided helps in making this a highly efficient interface, especially in the context of I/O intensive applications like web servers and databases. Our results show the potential performance advantages possible by exploiting this asynchrony to overlap I/O latency with useful processing. In effect, asynchrony helps hide the effect of disk latency since a carefully programmed application never needs to wait on I/O. This will have a significant impact on the throughput of such applications. Another useful flexibility given to the application in BAIO is the option to choose whether or not to use the global buffer cache of the kernel. Kernel filesystem always use the buffer cache and rawdisk I/O always bypasses it. Allowing the application decide on this on a per-request basis is bound to be a useful feature.

References

[1] The Linux kernel source code www.kernel.org

[2] P.Cao, E.W. Felten, and K.Li, Implementation and performance of application-controlled file caching. In Proceedings of the First Symposium on Operating Systems Design and Implementation, pages 165-178, November 1994

[3] Pei Cao, Edward W. Felten, Anna R. Karlin and Kai Li; Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling; ACM Trans. Comput. Syst. 14, 4 (Nov. 1996), Pages 311 - 343

[4] D. R. Engler, M. F. Kaashoek and J. O'Toole; Exokernel: an operating system architecture for application-level resource management; Proceedings of the fifteenth ACM symposium on Operating systems principles, 1995, Pages 251 - 266

[5] P.Brinch Hansen. The nucleus of a multiprogramming system. Communications of the ACM, 13 (4): 238-241, April 1970

[6] POSIX Asynchronous I/O URL: http://oss.sgi.com/projects/kaio/

[7] M.Stonebraker. Operating system support for database management. Communications of the ACM, 24(7); 4120418, July 1981.

[8] The textbook of the Linux kernel http://www.linuxdoc.org/LDP/tlk/tlk.html

[9] The Linux Kernel Hackers' Guide http://khg.redhat.com/HyperNews/get/khg.html