January 1997

Adaptive prefetching for device-independent file I

Dan Revel

Dylan McNamee

David Steere

Jonathan Walpole

Adaptive Prefetching for Device-Independent File I/O

Dan Revel, Dylan McNamee, David Steere, and Jonathan Walpole

{revel,dylan,dcs,walpole}@cse.ogi.edu

Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
20000 NW Walker Road, PO Box 91000
Portland, OR 97291-1000

Abstract

Device independent I/O has been a holy grail to OS designers since the early days of UNIX. Unfortunately, existing OS's fall short of this goal for applications that rely on very predictable I/O latency, such as multimedia players. Although techniques such as caching and sequential read-ahead can help by eliminating I/O latency in some cases, these same mechanisms can increase latency or add substantial jitter in others. In this paper we propose a new mechanism for achieving device-independent I/O − adaptive prefetching using application-supplied hints of access patterns. Adaptive prefetching actively monitors device performance and dynamically adjusts the amount of prefetching. Our experiments show device independence can be achieved: a Berkeley MPEG player sees the same latency when reading data from local disk or NFS. Moreover, our approach reduces jitter by a factor of 40 over standard techniques.

1) Introduction

Providing a device independent interface to resources has been a central feature of operating systems. For example, since the early days of Unix, application developers have enjoyed a simple and consistent, I/O independent, programming model provided by the file system abstraction. Programs handle data in files using a standard set of operations, such as open, read, write, seek, and close. In turn, the operating system translates these operations into device specific actions. The file system abstraction has proven to be a powerful tool, making it easy to write programs that store and access data on a wide variety of storage devices. However, current operating systems do not shield applications from variations in storage access latency due either to differences in hardware or variations in system load.

Applications that are sensitive to I/O latency and jitter, such as multimedia presentations are not provided with device independent I/O by existing operating systems. They must still explicitly consider timing issues, including disk seek latency and operating system queuing delays. They are either tuned to a specific set of device characteristics, thus limiting application portability, or they adapt their I/O behavior at run time at the cost of increased programming complexity.

In this paper we describe how device timing independent I/O can be provided to multimedia applications by combining application information, run-time monitoring and adaptation. We have implemented a user-level library which accepts application hints about future file accesses and actively manages the buffer cache. Application directed adaptive prefetching succeeds in hiding I/O latency in cases where prefetching based on a sequential readahead heuristic fails. Further, we maintain a prefetching window

This research is partially supported by DARPA grant N00014-94-1-0845, DARPA contract F19628-95-C-0193, NSF grant CCR-9224375, and grants from Hewlett-Packard, Intel and Tektronix.

and release data outside the window in order to limit the buffer cache flooding by large multimedia data streams.

This paper is organized as follows. Section 2 surveys related work in the areas of I/O prefetching and adaptive multimedia applications. Section 3 describes why I/O latency is a concern for multimedia and how current operating systems fail to meet the I/O latency requirements of multimedia applications. Section 4 presents our design and implementation of application directed adaptive prefetching. Section 5 describes the results of our experiment modifying the Berkeley MPEG player (mpeg_play) to use application directed adaptive prefetching. Section 6 presents our conclusions and provides directions for future research.

2) Related Work

Prefetching

I/O latency is a well known problem for storage hierarchies. System designers have used two basic techniques to address latency: prefetching and caching. Many researchers have noted that there is a strong interaction between prefetching and caching. This is not surprising since both are concerned with managing the same system resource: the file system buffer cache.

The prevalent operating system approach to buffer cache management has been to apply a sequential readahead policy for prefetching decisions [McKusick84,Freitag71] and a least recently used (LRU) cache replacement policy. These policies are easy to implement and have been shown to provide good performance for many applications.

Unfortunately, for some applications sequential readahead and LRU cache replacement provide poor performance. Database researchers pointed out the shortcomings of operating system buffer cache management policies more than fifteen years ago [Stonebraker81]. There is a body of research on how to improve database buffer cache management by using database and query specific information to select and tune buffer cache management policies for better performance [Chou85,Jauhari90].

The operating systems research community has also explored using application knowledge to make informed prefetching and caching decisions [Cao95,Patterson95,Kimbrel96,Mowry96]. Notably, Patterson’s Transparent Informed Prefetching (TIP) [Patterson95] uses application hints supplied at run time to make prefetching and caching decisions with the goal of increasing I/O throughput by exploiting the hardware parallelism of disk arrays. In contrast, Mowry reduces the I/O latency for out-of-core scientific applications by using compile-time analysis to insert prefetch and release calls into an application to perform explicit cache management [Mowry96].

In contrast to previous work in prefetching, our work combines application hints with run time performance monitoring and feedback. This mechanism allows our cache management decisions to adapt dynamically to storage devices with differing performance characteristics and also to varying workloads on a shared system.

Multimedia storage systems

Continuous media applications operate on large amounts of data and need to be able to access that data at consistent rates. For example, the MPEG-1 standard compresses video sequences into bitstreams that require a transfer rate of about 1.5 Mbps. Multimedia storage servers are designed to provide consistent rate access to continuous media data. Gemmell presents a survey of architectures and algorithms used to design multimedia storage servers [Gemmell95]. In general these servers are dedicated systems and use admission control to provide rate guarantees to their clients. The Fellini multimedia storage server proposes to support both continuous media and conventional data accesses [Martin]

The responsiveness of VCR-like functions, such as fast-forward and rewind, is particularly sensitive to I/O latency. Ozden's design for a Video-on-Demand server presents a scheme for a window of cached data around a playback point in order to support these functions [Ozden96].

Staehli and Maier [Staehli93,Maier93] have observed that storage systems may exploit multimedia content specifications to provide constrained latency storage access. In order to meet application requirements it is important not only to know what data is wanted, but also when that data is needed. This is especially true in multimedia where timing can be critical. Data that arrives too late may not be usable, and data that arrives too early consumes extra space in the buffer cache.

Adaptive applications

Because multimedia applications must run over the Internet where resource availability can vary greatly adaptivity is rapidly becoming a standard feature of networked multimedia applications. For example, McCanne's video conferencing tool, vic [McCanne95], uses the RTP multimedia transport protocol to monitor and adjust its bandwidth utilization.

The Quasar networked video player uses feedback both to adjust bandwidth utilization and to maintain client-server synchronization [Cen95,Koster96]. However, feedback based dynamic adaptation is not a simple task. Adapting too slowly means that variations in resource availability are not tracked wells, but adapting too quickly may result in overreaction or oscillation. The Quasar player uses the Streaming Control Protocol (SCP) and a toolkit of composable feedback modules to provide application-layer adaptivity.

Feedback and adaptation have been shown to be highly effective at adjusting application behaviors in response to variations in available network resources. We extend this work by applying it to hiding I/O latency for accessing data on secondary storage devices.

3) Motivation

This section explains how I/O performance affects multimedia applications. We begin by examining how I/O latency can limit the rate at which continuous media data is be presented. Next, we look at the techniques used by current operating systems and applications to hide I/O latency and we see how they may fail for multimedia applications. Finally, we present our approach to hiding I/O latency and discuss how it improves on current techniques.

I/O latency is the problem

Steadily increasing processor speeds have enabled a new generation of multimedia computing systems. These systems can manipulate and present continuous media data, such as digitally recorded audio and video, in real-time. This high volume data must often be streamed into memory from local or remote file systems on secondary storage devices. Consequently, I/O performance is often the bottleneck for multimedia applications.

The problem of I/O performance for multimedia applications is caused by the fact that the bandwidth and latency characteristics of disk drives have not kept pace with the growth of processor speeds, nor are they likely to in the future. RAID devices, redundant arrays of inexpensive disks, provide sufficient bandwidth by exploiting I/O parallelism. Even so, real-time constraints will continue to make multimedia applications sensitive to I/O latency when accessing stored data.

Typically, a demand-fetch paradigm is used to handle access to file systems on secondary storage: an application demands some data and then waits while the operating system fetches it from storage. This paradigm is unsuitable for multimedia applications. Consider, for example, a digital video recording consisting of a series of frames, averaging 8 KB in size, to be presented at a rate of 30 frames per second. (Note that this is low resolution video, 352x240 pixels.) Our example presents an I/O bandwidth demand of 240 KB per second, well within the capacity of current systems. In order to present this video it is necessary to read an average of one frame every 33 milliseconds. But, disk seek latency and queuing delays caused by competing I/O requests can easily result in more than 33 milliseconds of latency. Late data degrades the quality of a multimedia presentation by either causing a gap or a delay. One solution is to prefetch data before its use.

Prefetching turns the demand-fetch paradigm around. I/O latency can be hidden by continuously fetching data into memory before it is read by the application. However, fetching data too soon can also cause problems. Just five seconds of our example video will fill over a megabyte of memory, the buffer cache can quickly fill with video data which is only read once. The ideal is to have prefetched data streaming into the buffer cache so that it is available ’just in time’ for its presentation, and then releasing the data from the buffer cache soon after it has been presented.

Operating system heuristics

Operating systems are supposed to insulate applications from the details of managing the storage hierarchy. For example, prefetching and caching are done by the system, transparently to applications, using simple experience-based heuristic predictions of application reference behaviors. Sequential prefetching, for example, will prefetch data when a sequential access pattern is detected. These heuristics are easy to implement and provide good performance and a simple programming model for many applications.

One problem with relying on heuristic prefetching for multimedia is that it is reactive. Prefetching does not begin until after a sequential access pattern is detected. There is inevitably a delay between the time when an application starts accessing data (or changes the rate or pattern of access) and when the system adjusts to the new behavior.

Consider a multimedia presentation which concatenates two clips consisting of sequences of video frames stored in separate files. Once playback of the first clip has begun, sequential prefetching will work to hide the latency of accessing subsequent frames from that clip. But there is no way for the operating system to predict from the application’s reference behavior that the second clip will also be needed. When it is time to play the second clip, the application experiences the full delay of fetching the first frame of that clip from storage.

Another problem with relying on operating system prefetching heuristics is that accesses are not necessarily sequential even within a single stream. The bandwidth and processing requirements of playing a multimedia stream often tax the capacity of either the I/O system or the CPU or both. To compensate, a multimedia application may decide to skip or drop frames. For example, although a video may be encoded and stored at a playback rate of 30 frames per second or higher, an application may choose to play every other frame in order to reduce the bandwidth requirements. In addition, users may want to fast-forward or fast-rewind the video for a variety of purposes, including editting and scanning.

Application work-arounds

When an application's behavior does not fit the operating system's predictive model, developers must make a choice: either accept poor performance or try to work-around the operating system. Since poor performance does not sell well in a competitive marketplace the choice is simple: work-around the operating system limitation to get the desired performance. Similar situations have been found by database developers, who have been able to exploit their detailed application knowledge to improve I/O performance by explicitly managing the buffer cache. Multimedia applications are also in a good position to improve the way their data is managed by the storage hierarchy since they can predict their own future reference behavior better than the operating system can.

Reconsidering the example of concatenated video clips, the application might read frames some amount of time in advance of when they will be displayed. This would allow the latency incurred when beginning a new clip to be overlapped with the playback from memory of the last frames of the preceding clip. Some operating systems even provide explicit asynchronous I/O primitives to facilitate this sort of programming.

This level of system support allows multimedia applications to address the associated problems of latency, synchronization, and resource allocation on an ad hoc basis. By prefetching explicitly, an application can take advantage of its specific knowledge of what data will be needed in the future and when it needs to be available. We see three problems, however, with direct application management of prefetching: device dependence, complex shared resource management and lack of control over resources.

First, application management eliminates the device independence provided by operating system abstractions of resources. Instead applications must manually control the timing of prefetch requests. As a result, developers tune current high performance multimedia applications for specific storage devices [Aref97].

Second, application management can produce poor resource allocation and scheduling decisions in a shared environment. Shared resource management can be done better by a single subsystem rather than a collection of applications working independently. Further, without device-specific information, applications may use excessive amount of memory due to overly aggressive prefetching, or they may suffer from poor performance due to under-prefetching and dynamic system behavior. Another possibility is that applications may become overly complex trying to track and adapt to current system load.

Third, applications do not have the system level control needed enforce their decisions. For example, an application may try to buffer prefetched data in virtual memory expecting them to remain available for low-latency access only to have the data paged out by the virtual memory system [McNamee96]. One alternative is to allow applications to pin virtual memory pages so they could not be paged out. In this case, however, resource sharing is defeated.

Our solution

Our solution was motivated by the observation that good prefetching decisions depend on two sources of information: 1) system resource availability and performance characteristics, and 2) application behavior and quality of service requirements. Operating systems have explicit knowledge and control of system resources and try to predict how applications will behave. Applications have explicit knowledge and control of their own behavior and try to predict how the operating system will behave. Each has only one side of the picture and is faced with the difficult task of trying to dynamically predict the other.

We fix this problem by doing application directed adaptive prefetching with the goal of improving I/O performance by actively managing the file system buffer cache. We combine application-specific knowledge with operating system resource information to hide I/O latency and jitter from applications. This will simplify the task of programming multimedia applications by allowing them to be written without having to explicitly consider underlying storage devices.

Our primary goal is to hide I/O latency and jitter for multimedia applications. A secondary goal is to enable multimedia applications to become good citizens on shared systems by preventing large streams of continuous media data from flushing other data out of the shared buffer cache unnecessarily.

4) Design

This section describes our system architecture for hiding I/O latency and jitter through prefetching. We begin by explaining our general approach to the problem, then present the theoretical framework we use to make prefetching decisions, and finally we present our software architecture for hiding I/O latency through application directed adaptive prefetching.

Application directed adaptive prefetching

Our goal is to hide I/O latency and jitter incurred when reading data from secondary storage. In essence we use prefetching to improve file system performance by having data available in the buffer cache in memory when requested by the application. Effective prefetching demands good decisions about what data to prefetch and when to prefetch it.

Multimedia applications are in a good position to provide information about what data they will need and when they will need that data. For example, an MPEG player can provide an index of frames and a display rate. This tells us what data to prefetch and allows us to determine a deadline for each frame. We can work backwards from these deadlines to determine when to issue prefetch requests.

Prefetching works ahead of an application loading data into the buffer cache. The amount of work ahead is called the prefetch depth. In order to completely hide I/O latency, data must be requested sufficiently far in advance to insulate the application from system queuing delays and disk latency. The amounts of queuing delay and disk latency may vary depending on system hardware and workload. By actively monitoring these values in a running system we can dynamically adapt the prefetch depth in order to meet application deadlines.

We can establish upper and lower bounds on the prefetch depth. The lower bound on prefetch depth is trivially zero in the case where there is no prefetching. The upper bound on prefetch depth is called the prefetch horizon, this is the point at which I/O latency is completely hidden and there is no additional benefit to prefetching more deeply.

Theoretical framework

Prefetching seeks to hide I/O latency by having data already loaded in the buffer cache when it is requested by an application. If the requested data is not in the buffer cache then the requesting application experiences a stall of duration T_{stall} while it waits for the data to be fetched from secondary storage. Demand fetching, in which there is no work-ahead, gives us a worst case for T_{stall}; in this case T_{stall} equals the sum of the I/O queuing delay, T_{queue}, and the disk access latency T_{disk}. In general, T_{stall} is a function of the prefetch depth, that is how far ahead in the data stream we are prefetching. In the case of demand-fetching the prefetch depth is zero, thus:

1) $T_{stall}(0) = T_{queue} + T_{disk}$.

In order to estimate T_{stall} for a particular prefetch depth n, we use the rate of I/O accesses provided by the application. The inverse of the rate of I/O accesses is T_{period}. This is the amount of time per frame that may be overlapped with the servicing of I/O requests. Now we can generalize equation 1:

$$2)\ T_{stall}(n) = \max(T_{queue}+T_{disk}-(n * T_{period}), 0)$$

Now we can calculate our prefetch horizon, $n_{horizon}$, by solving equation (2) for $T_{stall}(n_{horizon}) = 0$. This gives us the prefetch depth at which I/O latency is completely hidden from the application:

$$3)\ n_{horizon} = (T_{queue} + T_{disk}) / T_{period}$$

We can further estimate queuing latency to be the product of the queue size and the disk access latency:

$$4)\ T_{queue} = Q_{size} * T_{disk}.$$

We can use these equations with values obtained from run time system monitoring to adaptively adjust the prefetch depth for multimedia data streams.

Software architecture

To test application directed adaptive prefetching we implemented it as a user level library. The library uses three new system calls. The first two, prefetch() and release() allow the buffer cache to be actively managed by requesting that blocks be loaded or unloaded from the cache respectively. The third, monitor(), provides access to information about the performance of the I/O subsystem including average disk access latency and current I/O queue size.

Our adaptive prefetcher accepts a index of frames to be read from the application and then proceeds in lock-step with the application by intercepting application read calls and using that time to monitor the I/O system and make prefetching decisions. Basically, we try to maintain a window of prefetched frames around the application’s current location in the data file. The size of window is varied adaptively based on the disk access latency and I/O queue size monitored in the system. Prefetches are issued to bring frames ahead of the application into memory and releases are used to release frames after they pass out of the prefetch window.

4) Experimental Results

In this section we examine the performance of an MPEG-1 video playback application with and without application directed adaptive prefetching. Since we are interested in cases where sequential readahead fails to hide I/O latency we looked at what happens when a multimedia stream is accessed in a striding pattern, for example when fast forwarding or when skipping frames to adapt to CPU or I/O bandwidth limitations.

MPEG-1 uses a combination of inter- and intra-frame coding in order to balance between the needs of compression and random access within a video sequence. Interframe encoding allows a number of techniques to achieve high compression be encoding only the differences between frames. As a result, interframe encoded frames depend on other frames in order to be decoded. In contrast, intraframe encoded frames are self-contained and provide convenient random access points within an MPEG bitstream. Within a bitstream frames are arranged in ’groups of pictures’ consisting of one intraframe encoded I-frame and zero or more interframe encoded P and B frames that depend on the I-frame and, possibly, each other. In our experiment we read and display only the I-frame from each group of pictures. As we have noted, this is a reasonable scenario and it produces a striding read pattern where sequential readahead does not work.

Next we describe the hardware and software used in our experiments. Then we describe our experiments and the metrics that were used. Finally, we present and discuss our results.

Equipment

We used the Berkeley MPEG video software decoder [Rowe93] (mpeg_play) version 2.2 to decode and display MPEG-1 video bitstreams in all of our experiments. We instrumented the mpeg_play application to measure CPU times using getrusage and an idle cycle counter read immediately before and after decoding and displaying the sequence of I-frames.

Our continuous media data file was an MPEG-1 bitstream recorded with an Hitachi MPEG camera (model MP-EG1A KIT ISA). The bitstream contained 128 groups of pictures each containing the following sequence of frame types: IBBPBBPBBPBBPBB. The 128 I-frames averaged 13710 bytes in size. Average sizes for P and B frames were 8495 and 5636 bytes, respectively. Thus our striding read pattern skipped 90340 bytes between each read. We used a separate index file that was created by parsing the bitstream before running our experiments. The size of the entire bitstream was 13.33 megabytes, with 1.76 megabytes, or 13%, containing I-frame data.

All of our experiments were run on 200 MHz Intel Pentium computers running the Linux 2.0.29 operating system. Our experiment required several modifications to Linux. Prefetch and release system calls were added to allow user-level cache management. Timestamps were added to I/O requests so that disk access latency could be monitored and made available to the adaptive prefetcher. And finally, we added an idle cycle counter as a way to measure I/O latency. We ran our tests on otherwise quiescent systems running only necessary background processes (X and nfsd). Under these conditions we interpret the measured idle time as roughly corresponding the amount of I/O stall time experience by the tested application.

Our local file system tests used Linux's ext2 file system and accessed data on a Quantum Fireball IDE hard drive. Using the Linux utility hdparm we measured buffered disk reads to take 5.7 MB/sec; reads of data already in the buffer cache were timed at 36 MB/sec. Our remote file system tests used an NFS file system also running on a 200 MHz Pentium Linux system. The client and server were connected using a dedicated 10Mbps ethernet.

Experiments

Our experiments consisted of using mpeg_play to display all the I-frames in an MPEG-1 bitstream. We tested three versions of mpeg_play. The first, mpeg_play, is the version from the Berkeley distribution. The second, indexed mpeg_play, is the version modified to use an index file that is read into memory before any measurements are made. And the third, prefetching mpeg_play, also uses an index file and is linked with a library that uses prefetch and release calls to do adaptive buffer cache management. Each of our versions of mpeg_play was tested on both local and remote file systems.

Results

Figure 1 shows the performance improvements achieved through using a index to read MPEG frames rather than reading and parsing the full bitstream. For each case we show a bar representing the average times observed over eighty runs. In each bar the top section is the amount of time the processor was idle, this corresponds to I/O stall time. The middle section of each bar is the time spent running in system mode - this includes time spent on copying data within the kernel. The bottom sections are the time spent in user mode decoding and displaying the MPEG video.

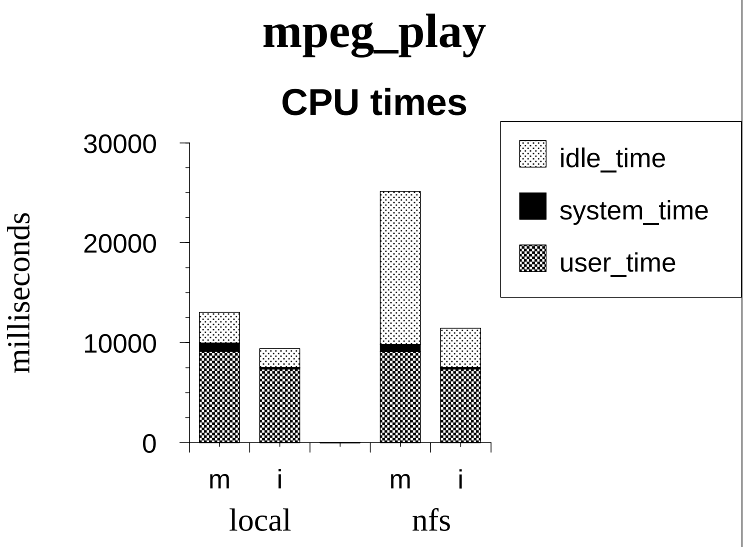

Figure 1: Performance improvement for mpeg_play using an index of frames. (m = no index, i = index)

We can see in Figure 1 that there is a dramatic performance leap between the original mpeg_play and the indexed version that reads frames based on an index. This result reflects the fact that the indexed version only reads 13% of the file, striding over the rest, whereas mpeg_play reads the entire file. It is interesting to note that in the indexed version experiences I/O stalls because its striding reads do not trigger Linux's sequential readahead. The mpeg_play version also experiences I/O stalls, but in this case the file is being read sequentially and Linux is doing readahead. These stalls occur because mpeg_play outruns Linux's readahead because it drops 87% of the data it reads without further processing. We thought this might be caused by mpeg_play's use of mmap, via the gcc libc implementation of fread, to access file data. Linux only does single page readahead for mmap. In contrast, Linux will readahead up to 36 pages in files accessed using read. We tried replacing mpeg_play's freads with reads and found that there was no change in performance even with the more aggressive readahead policy.

There is also a significant improvement in performance between the indexed and the prefetching versions of mpeg_play. Figure 2 shows this comparison, here both versions are handling the same number of bytes, but the indexed version spends more time stalled waiting for I/O because Linux's sequential readahead fails to recognize its striding pattern of reads. In the local disk case there was a 10% performance improvement for the prefetching application. In the NFS case, where I/O stall times were greater due to the higher latency of accessing the network server, the prefetching application showed a 20% performance improvement.

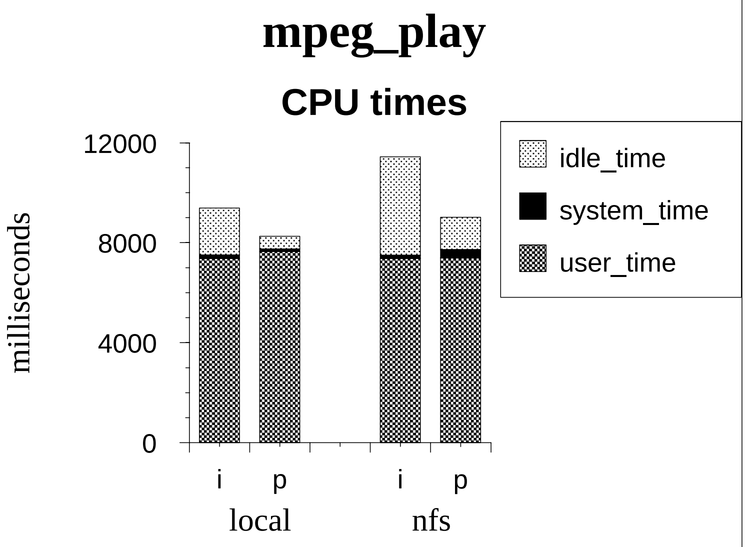

Figure 2: Performance improvement for mpeg_play using adaptive prefetching. (i = index, p = prefetch)

We still observed some idle time with adaptive prefetching because our prefetches are not completely asynchronous. On local file systems they stall while accessing file system metadata, inode block maps, and on remote file systems there are stalls in the network drivers. These delays can be reduced in the future by prefetching the metadata and by moving the prefetching activity from a user library to a seperate process.

Next we compare the read latencies observed by the indexed and the prefetching applications on single runs. Figure 3 and 4 show this comparison for reading data from a local hard drive and from a remote server. These runs are representative of the eighty runs we recorded. We have chosen to present single runs to give a sense of the variation in latencies that occur. In both the local and the remote cases, the read latencies for prefetching version averaged 0.3 milliseconds, this reflects the system call and data copying overhead. In contrast, the latencies for the indexed, non-prefetching, version reflect I/O stall times which are hidden by adaptively prefetching. For the local case, without prefetching, latencies averaged 14.7 milliseconds with a maximum of 187 milliseconds. For the corresponding remote case latencies averaged 31 milliseconds with a maximum of 53 milliseconds.

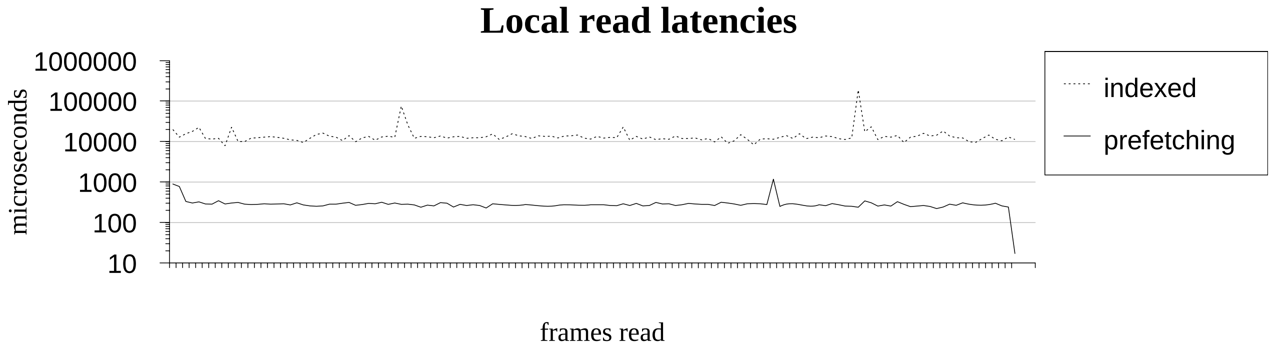

Figure 3: Comparing local read latencies for indexed and prefetching versions of mpeg_play (logarithmic scale)

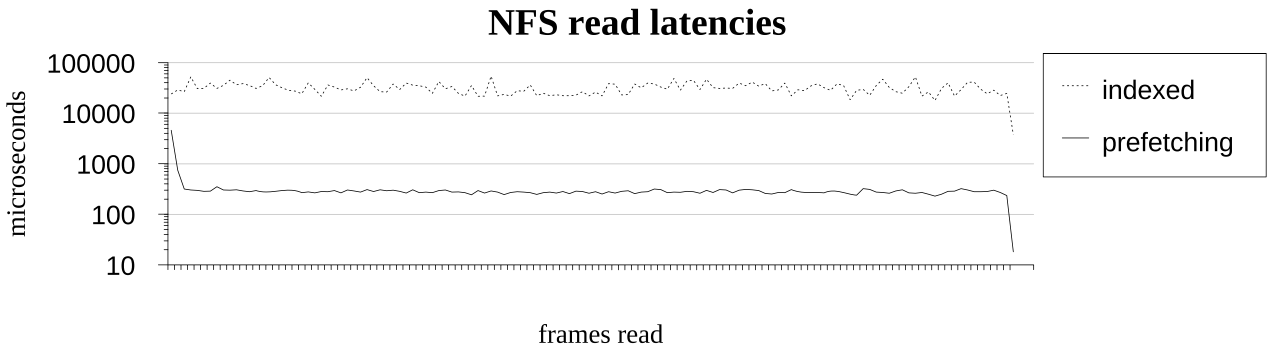

Figure 4: Comparing NFS read latencies for indexed and prefetching versions of mpeg_play (logarithmic scale)

In Figure 5 we show how we use a prefetching window to minimize the footprint of multimedia streams in the buffer cache. The strategy is simple and very successful: we leave frames in the cache after they have been read by the application for the same amount of time as we are prefetching in advance of the application. After this time the frames are explicitly release so that the buffers they occupy become available for reuse. By maintaining a window we ensure that should the application reverse directions, as may happen for example in the case of video editting, it is possible to proceed in the opposite direction without stalling to wait for I/O.

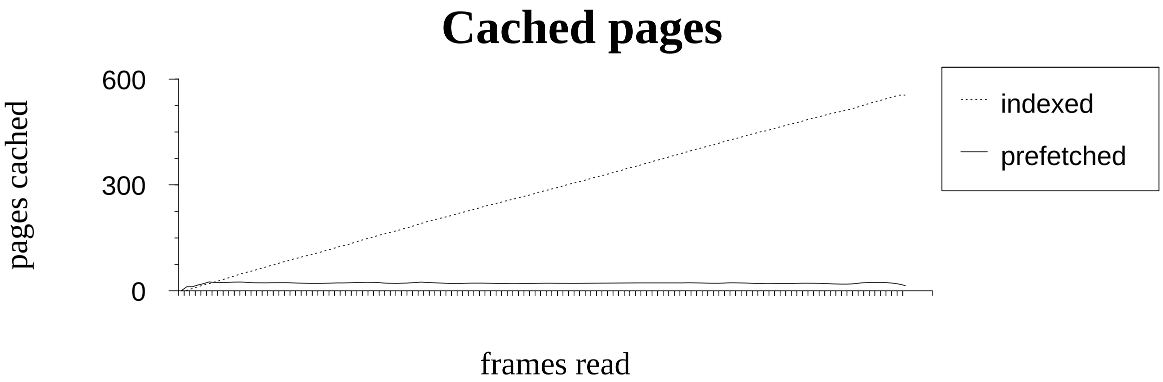

Figure 5: Comparison of buffer cache usage between LRU caching (indexed) and adaptively managed (prefetched)

5) Conclusions

In this paper we have demonstrated that it is possible to use adaptive prefetching to provide device independent file I/O. We can hide I/O latency and jitter from multimedia applications by using a combination of application information, runtime monitoring and adaptation. Our application directed adaptive prefetching works even in cases sequential readahead prefetching fails. In addition, we have shown that it is possible for multimedia applications to be *good citizens* and achieve good performance without flooding the buffer cache with continuous media data.

As CPU speeds continue to increase relative to secondary storage access latencies it will become increasingly important to prefetch data. By prefetching filesystem metadata as well as file data and by moving our prefetching operations from a user library to a seperate process we expect to be able to fully hide storage access latencies from applications.

The work described in this paper is an encouraging first step. We are currently extending our research in several directions. The prefetch, release, and monitor system calls give us a workable low-level interface on top of which we may construct adaptive prefetchers. The next step is to improve the application programming interface. We are also investigating how our system can implement and use admission control and I/O bandwidth reservations.

References

[Aref97] W.G. Aref, I. Kamel, T.N. Niranjan and S. Ghandeharizadeh. Disk Sceduling for Displaying and Recording Video in Non-Linear News Editing Systems. Proceedings of Multimedia Computing and Networking 1997. SPIE Proceedings Vol. 3020, San Jose, February 1997.

[Cao95] Pei Cao, Edward W. Felton, Anna Karlin, and Kai Li. A Study of Integrated Prefetching and Caching Strategies. Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1995.

[Cen95] Shanwei Cen, Calton Pu, Richard Staehli, and Jonathan Walpole. A Distributed Real-Time MPEG Video Audio Player. Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video, LNCS, v. 1018, Springer-Verlag, pages 142-153, 1995.

[Chou85] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies. Proceedings of the 11th VLDB Conference, Stockholm, Sweden, pages 127-141, August 1985.

[Freitag71] R. J. Freitag and E. I. Organisk. The Multics Input/Output System. Proceedings of the 3rd Symposium on Operating System Principles, pages 35-41, 1971.

[Gemmell95] D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. Venkat Rangan, and L. Rowe. Multimedia Storage Servers: A Tutorial and Survey. *IEEE Computer*, Vol. 28, No. 5, pages 40-49, May 1995.

[Jauhari90] Rajiv Jauhari, Michael J. Carey, and Miron Livny. Priority-Hints: An Algorithm for Priority-Based Buffer Management. Proceedings of the 16th VLDB Conference, Brisbane, Australia, pages 708-721, August 1990.

[Kimbrel96] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felton, Garth A. Gibson, Anna R. Karlin, and Kai Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. Proceedings of the 2nd Symposium on Operating Systems Design and Implementation, Seattle, Washington, pages 19-34, October 1996.

[Koster96] Rainer Koster. Design of a Mulitmedia Player with Advanced QoS Control. Master’s thesis, Oregon Graduate Institute of Science and Technology, Portland, Oregon, 1996.

[Maier93] D. Maier, J. Walpole and R. Staehli, Storage System Architectures for Continuous Media Data. In FODO ’93 Proceedings, LNCS, v. 730, Springer-Verlag, pp. 1-18, 1993.

[Martin] Martin, C., P. S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz. The Fellini Multimedia Storage Server. Multimedia Information Storage and Management, Editor -- S. M. Chung, Kluwer Academic Publishers, to appear.

[McCanne95] McCanne, S., and Jacobson, V., vic: A Flexible Framework for Packet Video. ACM Multimedia, November 1995, San Francisco, CA, pp. 511-522.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. ACM Transactions on Computer Systems, 2(3):181-197, August 1984.

[McNamee96] Dylan James McNamee. Virtual Memory Alternatives for Transaction Buffer Management in a Single-Level Store. Phd. thesis, Department of Computer Science, University of Washington, 1996.

[Mowry96] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. Proceedings of the 2nd Symposium on Operating Systems Design and Implementation, Seattle, Washington, pages 3-17, October 1996.

[Ozden96] Ozden, B., R. Rastogi, and A. Silberschatz. On the Design of a Low-Cost Video-on-Demand Storage System. Multimedia Systems Journal, February 1996.

[Patterson95] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. Proceedings of the Fifteeth ACM Symposium on Operating System Principles, pages 79-95, December 1995.

[Rowe93] L.A. Rowe, K. Patel and B.C. Smith, "Performance of a Software MPEG Video Decoder," Proc. ACM Multimedia 93, Anaheim, CA, August 1993

[Staehli93] Richard Staehli and Jonathan Walpole. Constrained-Latency Storage Access. IEEE Computer, Vol. 26, No. 3, pages 44-53, March 1993.

[Stonebraker81] Michael Stonebraker. Operating System Support for Database Management. Communications of the ACM, Vol. 24, No. 7, pages 412-418, July 1981.