1-1998

# Adaptive Prefetching for Device-Independent File I/O

Dan Revel

Dylan McNamee

David Steere

Jonathan Walpole
*Portland State University*



Citation Details

Dan Revel, Dylan McNamee, David Steere, and Jonathan Walpole, "Adaptive Prefetching for Device Independent File I/O," In Proc. SPIE 3310, Multimedia Computing and Networking 1998, 139 (December 29, 1997); doi:10.1117/12.298416

# 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 operating system designers since the early days of UNIX. Unfortunately, existing operating systems fall short of this goal for multimedia applications. Techniques such as caching and sequential read-ahead can help mask I/O latency in some cases, but in others they increase latency and add substantial jitter. Multimedia applications, such as video players, are sensitive to vagaries in performance since I/O latency and jitter affect the quality of presentation. Our solution uses adaptive prefetching to reduce both latency and jitter. Applications submit file access plans to the prefetcher, which then generates I/O requests to the operating system and manages the buffer cache to isolate the application from variations in device performance. Our experiments show device independence can be achieved: an MPEG video player sees the same latency when reading from a local disk or an NFS server. Moreover, our approach reduces jitter substantially.

**Keywords:** multimedia, prefetching, adaptive, I/O

## 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, device independent, programming model for I/O provided by the file system abstraction. Programs use a standard set of operations to handle data in files. 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, multimedia applications have placed demands on existing file systems that these systems were not designed to support.

Multimedia applications are sensitive to vagaries in performance since I/O latency and jitter affect the quality of presentation. These applications access large amounts of data on secondary storage through the file system interface. Because of their timing requirements these applications do not enjoy device independent I/O. Instead, they must explicitly consider timing issues, including disk seek latency and operating system queuing delays. Either applications are 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 a new architecture for application-directed prefetching and buffer management that provides device timing-independent I/O to multimedia applications. Device independence is acheived by combining application information, run-time monitoring and adaptation. We have implemented low-level controls for buffer cache management and a user-level library which accepts application hints about future file accesses and actively manages the buffer cache. Our library succeeds in hiding I/O latency in cases where sequential readahead fails. The library also limits the buffer cache footprint of multimedia streams by releasing data from the cache when it is no longer needed by the application and reusing the buffers that contained the data. We expand on our earlier work[1] through architectural and interface advances and we present new experimental results.

This project was supported in part by DARPA contracts/grants N66001-97-C-8522, N66001-97-C-8523, and F19628-95-C-0193, and by Tektronix, Inc. and Intel Corporation.

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 an MPEG video player to use our cache management library. Section 6 presents our conclusions and provides directions for future research.

## 2. RELATED WORK

This paper presents a new architecture for managing prefetching directed by adaptive applications. Related work includes multimedia storage systems that use admission control to ensure adequate performance. Other related work includes the adaptive applications our architecture is intended to support as well as other methods of managing prefetching.

### 2.1. 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.[2] In general these servers are dedicated systems and use admission control to provide rate guarantees to their clients. The Fellini multimedia storage server supports both continuous media and conventional data accesses[3] by using admission control to guarantee service levels. In contrast, our system is intended to support adaptive multimedia applications that can adjust their demands in response changes in resource availability.

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.[4]

Staehli *et al*[5] and Maier *et al*[6] 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.

### 2.2. Adaptive applications

Adaptivity is rapidly becoming a standard feature of networked multimedia applications that must run over the Internet where bandwidth availability can vary greatly. Video applications, for example, can adjust their bandwidth requirements by adding or dropping frames, switching resolutions or changing compression schemes. Applications can use protocols such as RTP[7] to monitor bandwidth availabilty and adjust their bandwidth utilization accordingly.[8]

The Quasar networked video player uses feedback both to adjust bandwidth utilization and to maintain client-server synchronization.[9,10] However, feedback based dynamic adaptation is not a simple task. Adapting too slowly means that variations in resource availability are not tracked well, 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[11] using a specialized server.

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.

## 2.3. 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. The prevalent operating system approach to buffer cache management has been to apply a sequential readahead policy for prefetching decisions[12,13] 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, the heuristic approach of sequential readahead and LRU cache replacement provides poor performance for applications with less predictable file access patterns or which exhibit poor temporal locality. Database researchers pointed out the shortcomings of operating system buffer cache management policies more than fifteen years ago.[14] 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.[15,16]

The operating systems research community has also explored using application knowledge to improve buffer cache management. Mowry *et al*[17] reduces the I/O latency for out-of-core scientific applications by using compile-time analysis to do low-level cache management by inserting prefetch and release calls into an application. A number of researchers have studied how to use application information at run-time.[17–20] Notably, Transparent Informed Prefetching (TIP)[19] shows how application hints about the timing, byte offsets and lengths of future read calls can be used to make prefetching and caching decisions. Similarly, Steere[21] proposes exploiting the semantics of search applications to perform opportunistic prefetching in wide-area distributed systems.

Numerous researchers have investigated exposing low-level page cache management to application control.[22–27] They have shown this approach to be feasible and to produce better performance in many cases. At a cost, however, of additional programming complexity. Additional software layers, such as libraries, are introduced to hide this complexity from applications.

There are a number of examples where the file system interface has been extended to fit the semantics of a particular application domain. Notably high level interfaces have been provided to describe fixed length strides and other regular patterns of data access.[28,29] The constant bitrate encoding of MPEG-1 bitstreams produces variable sized frames which cannot be easily captured as a regular pattern.

# 3. MOTIVATION

This section provides an overview of how operating systems can provide better I/O support to multimedia applications. We begin by looking at how I/O performance affects multimedia applications: variations in latency reduce the quality of multimedia presentations. 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.

## 3.1. 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 is often 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 because the bandwidth and latency characteristics of disk drives have not kept pace with 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. Other devices, such as CD-ROMs and wireless networks provide considerably lower bandwith and higher latency I/O. In order to function in this increasingly diverse environment, applications are forced to accomodate a wide range of bandwidth, latency and jitter variations.

Typically, a demand-fetch paradigm is used to handle access to file systems on secondary storage: an application demands some data and 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. To approach this ideal a prefetcher must know in advance what data will be needed and when it needs to be available.

## 3.2. 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.

### 3.2.1. Application adaptivity defeats OS heuristics

The big problem with relying on operating system cache management heuristics is that the file accesses of an adaptive multimedia application may not be 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, an adaptive 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. In all these cases files are accessed non-sequentially and the operating system heuristics are of no help, they may even be a hinderence if they fetch data in anticipation of a sequential access that never happens.

### 3.2.2. 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 operating system limitations 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 their own buffers.[15,16] 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 from each clip 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. Most operating systems 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,[30] resulting in reduced portability and increased complexity.

Second, application management can result in poor resource allocation and scheduling decisions in a shared environment. A single subsystem can manage shared resources more efficiently than a collection of applications working independently. Further, without device-specific information, expected latency and jitter, applications may use an 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 only to have the data paged out by the virtual memory system. 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.

### 3.3. Our solution

Our solution is 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 through application directed adaptive prefetching. Our goal is to improve I/O performance by actively managing the file system buffer cache, but we do not require applications to assume the responsibility of managing the buffer cache directly. This would complicate the task of application programming and would be highly error prone. Instead we provide a layer of system software, in the form of a library, which maps information provided by applications at run-time into the buffer cache management operations needed to hide I/O latency and jitter. This simplifies the task of programming adaptive multimedia applications by allowing them to simply declare their requirements and their anticipated behaviors.

## 4. DESIGN AND IMPLEMENTATION

This section describes how we provide latency and jitter sensitive multimedia applications with device independent I/O. We begin by explaining our general approach to the problem, then we present our software architecture for hiding I/O latency through application directed adaptive prefetching, and finally we descibe our implementation.

### 4.1. Application directed adaptive prefetching

Our primary goal is to provide I/O device independence for multimedia applications. We use prefetching to provide file access with low latency and by having data available in memory in the buffer cache when requested by the application. To be effective the system needs to make 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 to be played 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. More complicated frame access patterns can be expressed in a microlanguage that is interpreted by the system to extract block accesses and deadlines.

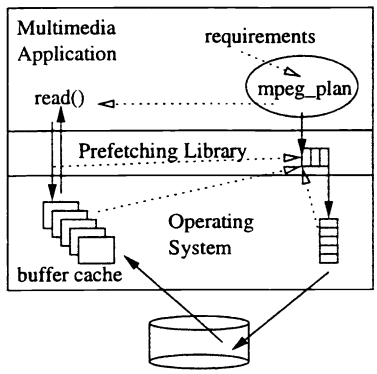


**Figure 1.** Buffer Management Architecture

Prefetching loads data into the buffer cache ahead of an application. 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 is the point at which worst case I/O latency is completely hidden and there is no additional benefit to prefetching more deeply.

### 4.2. Software architecture

We expose a set of low-level primitives to allow the implementation of alternate buffer cache management policies better suited to the needs of adaptive multimedia applications. Rather than have these applications implement new policies directly we have provided a middleware layer that takes file access plans and requirements from applications and manages the buffer cache accordingly.

Figure 1 shows our software architecture. We introduce a layer of middleware that provides adaptive multimedia applications with improved buffer cache management.

A set of low-level primitives is exposed by the operating system in order to allow alternate buffer management policies to be implemented at the user-level. The contents of the buffer cache are controlled by explicitly loading and unloading pages. In addition system conditions, such as average and worst-case I/O latencies and queue depths, are monitored and made available for inspection.

The middleware uses an application provided access plan to determine what the contents of the buffer cache should be at any given time. Monitored system conditions are used to make informed decisions about when pages need to be loaded into the cache and when they may be released. The conjunction of application provided information and system monitoring and control allows the middleware to make and enforce good cache management decisions.

### 4.3. Implementation

We chose to implement our buffer management middleware as a library. At the highest level a mapping function is provided to assist applications in translating their requirements into an access plan. Access plans are used by thebuffer manager to decide when pages need to be loaded and unloaded from the cache. Applications use the library interface to start and stop the buffer manager. The buffer manager uses low-level primitives exported by the OS kernel to monitor system conditions and control the contents of the cache. Table 1 shows the APIs for our buffer management library.

| Mapping Function | Library interface | Low-level Primitives |
|---|---|---|
| mpeg_plan | pf_init<br>pf_start<br>pf_lseek<br>pf_read<br>pf_stop | pf_ctl<br>pf_load<br>pf_unload |

**Table 1.** Buffer Management APIs

### 4.3.1. MPEG Access Plans

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.

An adaptive application may selectively skip frames to adjust to limited I/O (or network) bandwidth. Because of the interframe dependencies in MPEG bitstreams some care is required when choosing which frames to skip. `mpeg_plan` is a mapping function that accepts an index for an MPEG bitstream and generates an access and playback plan that conforms to a specified bandwidth. A continuous range of bandwidths is supported. High and low bandwidth access patterns might trivially be supported by coping the frames in a low bandwidth pattern to a new file allowing sequential access, however this solution is infeasible where a range of adaptation patterns is possible.

### 4.3.2. Library Interface

Applications use `pf_init` to inform the library about an access plan. `pf_init` sets up data structures to track and manage the access stream, if necessary it prefetches the beginning of the stream and then it waits for the application to signal it to start.

`pf_start` is used to signal the prefetcher that the application is starting to use an access stream that has already been set up using `pf_init`. This allows applications to synchronize the starts of several streams or to set up several alternative streams and to select which one will be used (as might happen with an interactive application).

Applications use `pf_lseek` and `pf_read` to access files. These allow the library to keep track of the application as it reads through the access stream. These functions pass calls through to lseek and read, respectively.

Streaming access is terminated using `pf_stop`. Currently, the behavior of an access stream is changed by stopping the current stream and starting a new one.

### 4.3.3. Low-level Primitives

We have exposed low-level buffer cache management to our middleware in the form of three system calls: `pf_ctl`, `pf_load`, and `pf_unload`. These calls provide the low-level primitives upon which a user-level buffer manager may be constructed.

`pf_ctl` is used to turn user-level buffer cache buffer management on for a given file descriptor. It does this by disabling the operating system's default policies and enabling the `pf_load` and `pf_unload` system calls. `pf_ctl` is also used to access information about the performance of the I/O subsystem including average disk access latency and current I/O queue size.

`pf_load` and `pf_unload` allow the buffer cache to be actively managed by requesting that pages be loaded or unloaded from the cache respectively. Loading and unloaded is done asynchronously. However Linux handles ext2 file meta-data synchronously so these calls may block while file offsets are being mapped to disk blocks. In order to isolate applications from this blocking our buffer manager runs in a seperate thread.

## 5. 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.

We have an adaptive multimedia player[31] that does the application work-arounds that we wish to obviate. We discovered that removing these work-arounds was too difficult (which enforces our claim that they add complexity). So, we decided to emulate an adaptive player using different access patterns by using a non-adaptive player to use the same patterns. In our experiments 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.

### 5.1. Equipment

We used the Berkeley MPEG video software decoder[32] version 2.2 to decode and display MPEG-1 video bitstreams in our experiments, hereafter referred to as the player. The player was modified to read MPEG frames using an index of their byte offsets and lengths within the source file. In addition, we instrumented the player to measure read call latency.

Our continuous media data file was an MPEG-1 bitstream recorded with an Hitachi MPEG camera (model MPEG1A KIT ISA). The bitstream contained 128 groups of pictures each containing the following sequence of frame types: IBBPBBPBBPBBPBB. The 132 I-frames averaged 13120 bytes in size. Average sizes for P and B frames were 6561 and 3260 bytes, respectively. Thus our striding read pattern skipped about 58844 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 9.03 megabytes, with 1.73 megabytes containing I-frame data.

All of our experiments were run on 233 MHz Intel Pentium computers running the Linux 2.0.30 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 SEAGATE Barracuda 4LP disk drive with an Ultra-SCSI Wide interface. Using the Linux utility hdparm we measured buffered disk reads to take 9 MB/sec; reads of data already in the buffer cache were timed at 50 MB/sec. We ran the NFS server for our remote file system tests on the same machine so that the disk drive and data layout remained the same. The client and server were connected using a dedicated 10Mbps Ethernet. The server used an Intel EtherExpress Pro 10/100 network card and the client used a Kingston NE2000 clone. The hardware and software configuration of the client machine was otherwise identical to that of the server.

### 5.2. Experiments

Our experiments consisted of using the player to display all the I-frames in an MPEG-1 bitstream. We measured the latency application read calls with and without our buffer management library accessing files on a local disk and on an NFS server. We also measured the total number of pages used in the buffer cache over time for each file read.

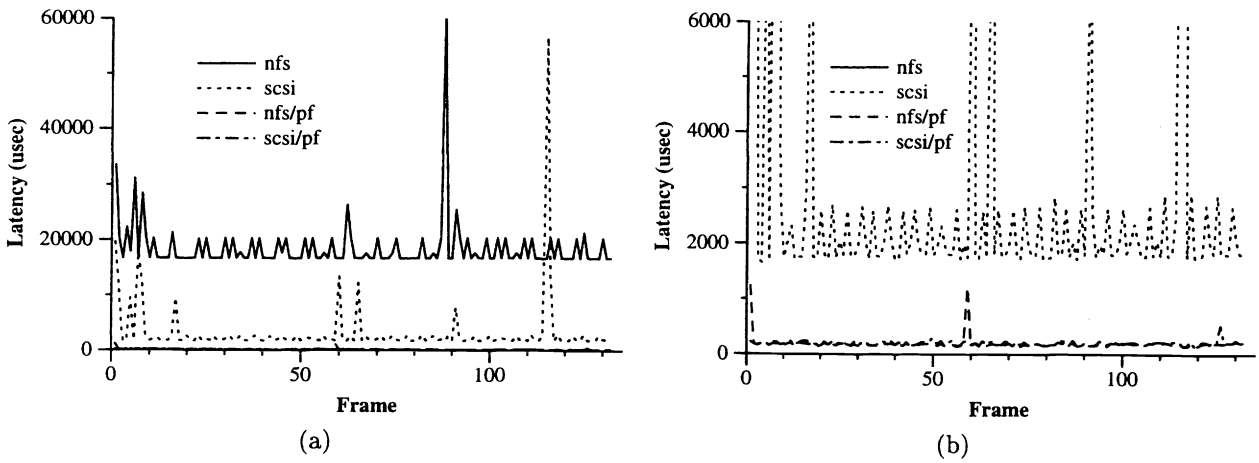


**Figure 2.** Read call latencies for non-prefetched and prefetched frames. (a) shows the difference in latency between NFS and SCSI devices. (b) is a detail showing the very low latency for reading prefetched frames.

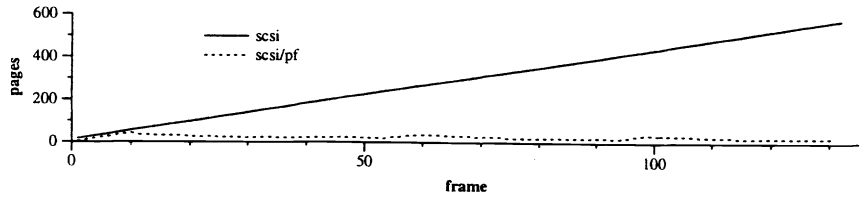


**Figure 3.** Cached pages

### 5.3. Results

Figure 2 (a) shows the read latencies observed for each frame read from files managed using Linux's default policies. As expected the NFS filesystem exhibited an order of magnitude greater latency than the local scsi disk, 20 ms vs. 2 ms. Both file systems displayed intermitent latencies of up to 60 ms. In contrast the latency for frames read from files managed by our buffer management library averaged about 200 microseconds, this is corresponds to the amount of time it takes to do a memory to memory copy of the data. Figure 2 (b) shows the same results, magified so that the read latencies for the prefetched cases are visible. Notice that there is almost no jitter for these cases.

In figure 3 we show that our buffer manager limits the memory footprint of multimedia data by unloading pages that are no longer needed. Operating system heuristics that detect a sequential access pattern produce similar results, but in this case Linux is unable to detect a sequential access because the application is striding through the data.

## 6. CONCLUSIONS

This paper has descibed an architecture and implementation for application directed adaptive prefetching. We have demonstrated that it is possible to use adaptive prefetching to provide device independent file I/O. We hide I/O latency and jitter from multimedia applications by using a combination of application information, runtime monitoring and adaptation. Our prefetching works, even in cases where sequential readahead fails.

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 working on a microlanguage to descibe application file access patterns. We are also investigating how our system can implement and use admission control and I/O bandwidth reservations.

## REFERENCES


1. D. Revel, C. Cowan, D. McNamee, C. Pu, and J. Walpole, "Predictable file access latency for multimedia," in *Proceedings of the Fifth IFIP International Workshop on Quality of Service*, pp. 401–404, (New York, NY), May 1997.
2. D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan, and L. Rowe, "Multimedia storage servers: A tutorial and survey," *Computer* **28**, pp. 40–49, Aug. 1995.
3. C. Martin, P. S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz, "The fellini multimedia storage server," to appear.
4. B. Ozden, R. Rastogi, and A. Silberschatz, "On the design of a low-cost video-on-demand storage system," *Multimedia Systems Journal* , Feb. 1996.
5. R. Staehli and J. Walpole, "Constrained-Latency Storage Access," *COMPUTER* **26**, pp. 44–53, Mar. 1993.
6. D. Maier, J. Walpole, and R. Staehli, "Storage system architectures for continuous media data," in *Foundations of Data Organization and Algorithms, FODO '93 Proceedings*, D. B. Lomet, ed., vol. 730 of *Lecture Notes in Computer Science*, pp. 1–18, Springer-Verlag, 1993.
7. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "Rtp: A transport protocol for real-time applications." Network Working Group Request for Comments: 1889, Jan. 1996.
8. I. Busse, B. Deffner, , and H. Schulzrinne, "Dynamic qos control of multimedia applications based on rtp," in *Proceedings of the First International Workshop on High Speed Networks and Open Distributed Platforms*, (St. Petersburg, Russia), June 1995.
9. S. Cen, C. Pu, R. Staehli, and J. Walpole, "A distributed real-time mpeg video audio player," in *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video, LNCS* **1018**, pp. 142–153, Springer-Verlag, 1995.
10. R. Koster, "Design of a mulitmedia player with advanced qos control," Master's thesis, Oregon Graduate Institute of Science and Technology, Portland, Oregon, 1996.
11. S. Cen, *A Software Feedback Toolkit and its Application In Adaptive Multimedia Systems.* PhD thesis, Oregon Graduate Institute of Science and Technology, Oct. 1996.
12. M. K. McKusick, W. Joy, S. Leffler, and R. Fabry, "A fast file system for UNIX," *Transactions on Computer Systems* **2**, pp. 181–197, Aug. 1984.
13. R. J. Freitag and E. I. Organik, "The multics input/output system," in *Proceedings of the 3rd Symposium on Operating System Principles*, pp. 35–41, 1971.
14. M. Stonebraker, "Operating system support for database management," *Communications of the ACM* **24**, pp. 412–418, July 1981.
15. H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies," in *Proceedings of the 11th VLDB Conference*, pp. 127–141, (Stockholm, Sweden), Aug. 1985.
16. R. Jauhari, M. J. Carey, and M. Livny, "Priority-hints: An algorithm for priority-based buffer management," in *Proceedings of the 16th VLDB Conference*, pp. 708–721, (Brisbane, Australia), Aug. 1990.
17. T. C. Mowry, A. K. Demke, and O. Krieger, "Automatic compiler-inserted I/O prefetching for out-of-core applications," in *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pp. 3–17, USENIX Association, Oct. 1996.
18. P. Cao, E. W. Felton, A. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Proc. ACM* , May 1995.
19. R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Dec. 1995.
20. T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felton, G. A. Gibson, A. R. Karlin, and K. Li, "A trace-driven comparison of algorithms for parallel prefetching and caching," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pp. 19–34, (Seattle, Washington), Oct. 1996.
21. D. C. Steere, "Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency," in *Proceedings of the 16th ACM Symposium on Operating System Principles*, Oct. 1997.

22. D. R. Engler, M. F. Kaashoek, and J. O. Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), ACM Operating Systems Review, SIGOPS* **29**(5), 1995.
23. 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 *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pp. 267–284, (Copper Mountain Resort, CO), December 1995.
24. C. H. Lee, M. C. Chen, and R. C. Chang, "HiPEC: High performance external virtual memory caching," in *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pp. 153–164, 1994.
25. D. McNamee and K. Armstrong, "Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies," in *Proceedings of the First USENIX Mach Symposium*, pp. 17–29, (Burlington, Vermont), October 1990.
26. K. Harty and D. Cheriton, "Application-controlled physical memory using external page-cache management," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 187–197, Oct. 1993.
27. P. Cao, E. W. Felten, and K. Li, "Application Controlled File Caching Policies," in *Proceedings of the USENIX Summer 1994 Technical Conference*, pp. 171–182, June 1994.
28. J. F. Karpovich, J. C. French, and A. S. Grimshaw, "High performance access to radio astronomy data: A case study," in *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, Sept. 1994.
29. N. Nieuwejaar and D. Kotz, "The galley parallel file system," in *Proceedings of the 1996 International Conference on Computing*, pp. 374–381, ACM, (New York), May25–28 1996.
30. W. Aref, I. Kamel, T. Niranjan, and S. Ghandeharizadeh, "Disk sceduling for displaying and recording video in non-linear news editing systems," in *Multimedia Computing and Networking, Proc. SPIE* **3020**, Feb. 1997.
31. J. Walpole, R. Koster, S. Cen, C. Cowan, D. Maier, D. McNamee, C. Pu, D. Steere, and L. Yu, "A player for adaptive mpeg video streaming over the internet," in *Proceedings of the 26th Applied Imagery Pattern Recognition Workshop AIPR-97*, SPIE, (Washington DC), Oct. 1997.
32. L. Rowe, K. Patel, and B. Smith, "Performance of a software mpeg video decoder," in *Proceedings of the First ACM International Conference on Multimedia (MULTIMEDIA '93)*, (Anaheim, CA), Aug. 1993.