with a picture group pattern of IBBPBBPBBPBB. (MPEG encodings contain three types of frames, I, P, and B, that vary in size and importance.) The pattern 100000000000 requests that only the I-frames be prefetched. The pattern 100100100100 requests both I-frames and P-frames. Patterns can be of arbitrary length. For example, 100100100100100100100000 requests all I-frames and five of every six P-frames.

Finally the fifth argument to `pf_start` is the rate at which the frames will be accessed. This is in frames per second with a negative number indicating that the frames will be read in a backwards direction.

If a feasible prefetching plan can be constructed then `pf_start` sets it into action and returns a descriptor. The application uses the descriptor to refer to the plan in the future to modify or stop it. It is possible to initiate several prefetching sequences for the same file.

`pf_start` fails and returns zero if the prefetcher determines that given available resources there is not a feasible prefetching plan that will satisfy the request.

`pf_modify()` changes an executing prefetching plan. This might be used, for example, when a user pushes the fast-forward button in a VCR application. The `frame` argument is taken to indicate the application's current position in accessing the data file and the revised prefetching plan will begin from that point. Pattern and rate have the same meaning as for the `pf_start` call.

`pf_stop` terminates the execution of a prefetching plan.

5 Related Work

Prefetching has been studied extensively in a number of different contexts.

Patterson, *et al*, advocate hints that disclose an application's future reference behavior and makes some suggestions about what the form of those hints might be. They also present a cost-benefit analysis for prefetching decisions [Patterson95]. Effective prefetching algorithms when full advance knowledge is available are described by Kimbrel, *et al* [Kimbrel96].

Maier, *et al*, present a storage system architecture for a multimedia database system which provides 'constrained-latency storage access' to continuous media data on high-latency storage devices [Maier93].

6 Conclusions and Future Work

We have described an architecture for flexible prefetching and we believe that it will be an important component of effective and device-independent multimedia systems. By implementing and using a prefetcher based on this architecture we hope to gain experience with the ways in which a prefetcher can interact with both applications and the underlying system.

In the future we plan to investigate how our architecture can be extended to support a distributed networked multimedia environment. We will also be investige a more expressive meta-interface that will allow applications to express their prefetching requests at a higher level (e.g. through content specifications or scripts).

References

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

[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", in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation,* pp 19-34, October 1996.

[Koster96] Rainer Koster. "Design of a Multimedia Player with Advanced QoS Control", Master's thesis, Department of Computer Science & Engineering, Oregon Graduate Institute of Science & Technology, 1996.

[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

[Maier93] David Maier, Jonathan Walpole and Richard Staehli. "Storage System Architectures for Continuous Media Data", in *Foundations of Data Organization and Algorithms, FODO '93 Proceedings, Lecture Notes in Computer Science*, Vol. 730, 1993, Springer-Verlag, Editor David B. Lomet, pp. 1-18.

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

even though the data is stored on a high latency secondary storage device.

The prefetcher executes a plan by issuing a stream of non-blocking load requests to the filesystem. The load requests have the effect of 'warming-up' the data in filesystem's buffer cache by fetching it from secondary storage, but the data is not made available to the application until it is explicitly read through the traditional filesystem interface.

The prefetcher uses the following equation to decide how far in advance to issue load requests in order to meet the application's stated requirements:

Prefetch_Depth = *I/O_Latency* * *Display_Rate* * ρ

I/O_Latency is an estimate of the amount of time which will elapse between when the request is issued and when the data is available in memory. *Display_Rate* is the rate of access disclosed by the application. ρ is a value >= 1 making *Prefetch_Depth* slightly larger to allow for fluctuations in actual I/O latency.

The prefetcher requires the assistance of the operating system in order to accurately estimate and predict current and future I/O latency. In turn, the prefetcher can assist with the operating system's efforts to minimize I/O costs by issuing load requests in batches that may, for example, be reordered by a disk scheduler.

4 Implementation

Our initial implementation provides the application API shown in Figure 2. This API is limited and focuses on the requirements of a particular multimedia player handling real-time MPEG video and audio streams [Koster96]. In the future, we plan to develop a more flexible and expressive API that will accept complex multimedia content specifications.

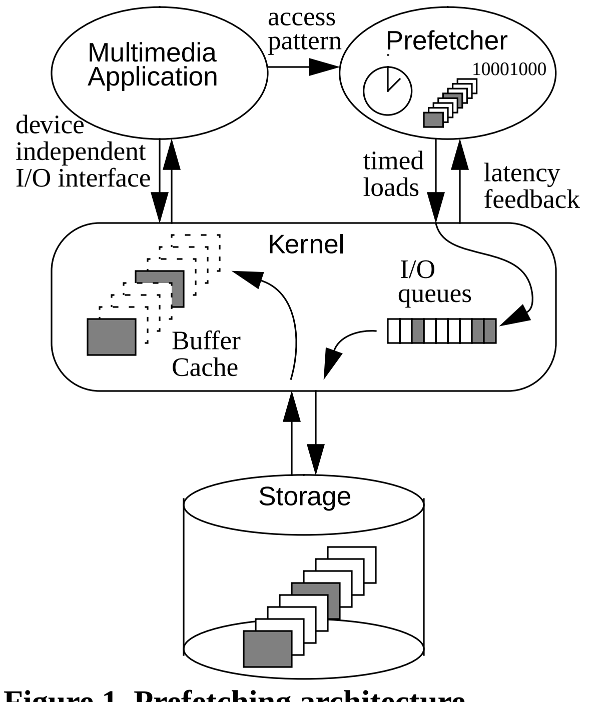

Figure 1. Prefetching architecture

```
typedef struct frame {
   off_t offset;
   size_t length;
} frame;

typedef struct pattern {
   int size;
   byte *bits
} pattern;
typedef int rate;

int pf_start(int fd, int n_frames,
    frame *frames, pattern *pattern,
    rate rate);

int pf_modify(int pd, int frame,
    pattern *pattern, rate rate);

int pf_stop(int pd);
```

Figure 2. Prefetcher meta-interface

The `pf_start()` call initiates a prefetching sequence by requesting the prefetcher to create and execute a prefetching plan. `pf_modify()` is used to change the access pattern and rate of an existing plan. `pf_stop()` terminates a currently executing plan.

The `pf_start` call takes five arguments. First, a file descriptor specifying the file containing the data to be prefetched. The application needs to open the data file before initiating a prefetching plan. Once a prefetching plan has been started the application simply uses the traditional `read()` call to access the data. Closing a file terminates any outstanding prefetching plans.

The second argument to `pf_start` is a count of the number of frames to be prefetched. This is used to keep the prefetcher from overrunning the end of a prefetch sequence. It is also used as a starting point for sequences that are accessed in a backwards order (see rate below). Note that 'frame' is used here to refer to a unit of application data, this might well be a video frame, an audio slice or some other object.

The third argument to `pf_start` is a vector describing the location of each frame in the data file and its size. This is basically an index. Note that the pattern and rate arguments are used to avoid having to generate a new index to describe each possible access pattern.

The fourth argument to `pf_start` is the pattern in which frames will be accessed. This is implemented as a bit vector representing a repeating pattern of selected frames. Consider a video player reading an MPEG video

2 Motivation

Multimedia applications require real-time access to large amounts of data on secondary storage (magnetic disks, optical disks, etc.). Many storage devices available today can sustain the bandwidth required by multimedia applications. The latency of accessing data on these devices, however, presents the very real problem that data may arrive too late. I/O latency can be hidden by prefetching data into memory before it is needed by the application. But, fetching data too soon can also cause problems by increasing the amount of memory needed to buffer data. The ideal is to have prefetched data streaming into memory so that it is available 'just in time' as it is needed by the application [Maier93]. We propose that the best way to approach this ideal is to base prefetching decisions on information from several sources:

- meta data describing framing and timing of the continuous media data,
- application specific playback rates and access patterns, and
- global resource allocation and usage information.

Many filesystems recognize when a file is being accessed sequentially and do heuristic prefetching [Freitag71][McKusick84]. Some even use stochastic techniques to adjust the depth of prefetching depending on the rate at which the data is being accessed. This approach takes advantage of an access pattern that can be easily inferred to provide the latency hiding benefits of prefetching, while limiting the amount of memory used to buffer prefetched data. The problem with using heuristic prefetching for multimedia applications is that it is *reactive*. There is inevitably a delay between the time when an application starts accessing data (or changes the rate at which it is being accessed) and when the system adjusts to the new behavior. Further, when data is accessed in a non-sequential pattern, for example fast-forward, then no prefetching at all will occur.

Without system support for prefetching, multimedia applications must address the associated problems of latency, synchronization, and resource allocation on an *ad hoc* basis. An application that handles prefetching explicitly can take advantage of specific knowledge of what data is likely to be needed in the future and the rate at which it needs to become available. This means that the application is no longer I/O device-independent, instead it must explicitly consider on which device the data is located and how long it will take to fetch it into memory. There are short-cuts which might be taken (e.g. greedy prefetching or estimating I/O latency) but these are at best inefficient and potentially unreliable.

The operating system cannot distinguish application prefetch requests from data requests. The operating system attempts to optimize I/O by scheduling data requests. When prefetch requests are intermingled, the operating system's schedule can be inefficient. Individual applications, however, possess valuable information for data prefetching, but they are ill-equipped to address global resource scheduling issues. Providing the operating system with knowledge of application needs allows the OS to make reasonable global resource allocation decisions [McNamee].

In our design we propose a meta-interface that allows an application to describe the pattern and timing of future data accesses to the prefetcher. After providing this description the application can proceed to use a traditional device independent interface to access its data with the promise from the system that the data will be in memory when it is needed.

The prefetcher translates the application's description into a scheduled stream of prefetch requests that satisfies the application's access requirements and makes efficient use of system resources. We expect that this will improve the performance of individual multimedia applications and also make them easier to write. Our goal is to define an architecture for prefetching, to construct a prefetcher, and to develop an understanding of the interactions between applications, the prefetcher and the operating system.

3 Architecture

Figure 1 shows our overall architecture for flexible prefetching. There are three basic components: the multimedia application, the prefetcher and the operating system. The multimedia application wants low-latency access to data on the storage managed by the operating system. To achieve low-latency access, the application tells the prefetcher what data it want to access and when it wants to be able to access it. The prefetcher then formulates and executes a prefetching plan that will meet the application's requirements. The operating system cooperates with the prefetcher by providing it with current latency and scheduling information.

The prefetcher provides applications with a meta-interface. We call the prefetcher's interface a meta-interface because it does not change the semantics of the filesystem's I/O interface, but rather it changes the way I/O requests are serviced. Specifically, an application uses the prefetcher's interface to describe the pattern and timing of its anticipated data accesses. In response, the prefetcher tries to make sure that the data will be in memory at the required time. With the information provided by the application, the prefetcher is able to provide the application with low-latency access to its data

An Architecture for Flexible Multimedia Prefetching

Dan Revel, Crispin Cowan, Dylan McNamee, Calton Pu and Jonathan Walpole
{*revel, crispin, dylan, calton, walpole*}*@cse.ogi.edu*

Department of Computer Science & Engineering
Oregon Graduate Institute of Science & Technology
20000 N.W. Walker Rd., P.O. Box 91000
Portland, OR 97291-1000

November 20, 1996

Abstract

Increasing CPU speed and I/O bandwidth have enabled the development of multimedia systems that handle continuous media data in real-time. Real time constraints of multimedia make the schedule with which data is delivered to main memory critical to application performance. Data must be prefetched from secondary storage into main memory in order to hide I/O latency.

We propose an architecture for flexible prefetching. In order to be effective the prefetcher must know in advance what data will be accessed and when it must be available in memory. In addition, it must have information about the system on which it is running: the amount of I/O latency and the availablity of I/O bandwidth and main memory.

Our prefetching architecture will simplify the task of programming multimedia applications and it will be useful for constructing complex distributed multimedia systems.

1 Introduction

Increasing CPU speed and I/O bandwidth have enabled the development of multimedia systems that handle continuous media data in real-time. The high volume of continuous media data relative to main memory size makes it necessary to store this data on secondary storage devices and to stream it into memory when it is needed. The real time constraints of multimedia make the schedule with which data is delivered to main memory critical to application performance. Data must be requested before it is needed in order to arrive in memory on time. This is the familiar concept of prefetching.

Prefetching hides the I/O latency of accessing secondary storage. Many filesystems prefetch data automatically for files that are being accessed sequentially. But, because continuous media data may be accessed non-sequentially (e.g. fast-forward), multimedia applications are currently forced to assume the responsibility for prefetching data explicitly. To correctly prefetch in a multimedia application entails taking into account the data access pattern and timing, as well as device specific attributes, such as latency and bandwidth. We feel making applications be explicitly responsible for their prefetching a bad idea for several reasons:

- it adds to the burden of programming multimedia applications,
- it eliminates the device independence of the I/O interface, and
- it places the responsibility for scheduling and resource allocation decisions on the application which does not have the information or control needed to make these decisions effectively.

We propose an architecture for prefetching in which a seperate flexible prefetching component is responsible for the scheduling and resource allocation decisions necessary to have data in memory when it is needed. In order to be effective the prefetcher must know in advance what data will be accessed and when it must be available in memory. In addition, it must have information about the system on which it is running: the amount of I/O latency and the availablity of I/O bandwidth and main memory. I/O latency and bandwidth determine how far in advance data must be requested. Finally, the availability of main memory determines how much prefetched data can be buffered in anticipation of future accesses by the application.

This project is supported in part by grants from DARPA and the National Science Foundation,and donations from Tektronix, Hewlett-Packard and the Portland Trail Blazers.