Synthetic Files: Enabling Low-latency File I/O for QoS-Adaptive Applications

Dylan McNamee, Dan Revel, Calton Pu, David Steere, Jonathan Walpole
Oregon Graduate Institute of Science and Technology
http://www.cse.ogi.edu/DISC/projects/quasar

Abstract

Files are a tried and true operating system abstraction. They present a simple byte-stream model of I/O that has proven intuitive for application programmers and efficient for operating system builders. However, current file systems do not provide good support for adaptive continuous media (CM) applications – an increasingly important class of applications that exhibit complex access patterns and are particularly sensitive to variations in I/O performance.

To address these problems we propose synthetic files. Synthetic files are specialized views of underlying regular files, and convert complex file access patterns into simple sequential synthetic file access patterns. Synthetic file construction can be viewed as a declarative meta-interface for I/O, enabling application-driven prefetching strategies that can hide device access latency even for applications with complex access patterns. Synthetic files can be realized dynamically, incrementally, or even optimistically. In this paper we outline a feedback-driven, incremental creation strategy that hides variations in device access latency for QoS-adaptive CM applications.

1 Introduction

Files are a tried and true operating system abstraction. They offer a clean and simple byte-stream model of I/O that is intuitive for application programmers and provides a degree of device-independence across different underlying physical storage devices. File system prefetching and caching mechanisms ensure that most file operations access memory rather than secondary storage, making file system performance not only fast, but predictable. However, because of the use of simple heuristics, such as sequential read-ahead, the effectiveness of these mechanisms in current systems depends heavily on file access patterns. Two implicit assumptions motivate the use of such heuristics: that access patterns are predominantly sequential - an apparently reasonable assumption given the serial byte-streaming model on which the file abstraction is based; and that buffer-cache misses, which incur the full latency of accessing secondary storage when the heuristic fails, are not a problem for applications. With the advent of QoS-adaptive continuous media applications – an increasingly important application class – neither of these assumptions is appropriate.

Since such applications manipulate continuous media data with real-time display requirements, they are particularly sensitive to the timing behavior of I/O processing. In order to preserve a real-world playout rate, they adapt to variations in resource-level throughput by dropping data, and consequently reduce the quality of the media rather than the rate at which it is delivered. For example, when insufficient bandwidth is available to a streaming video player it drops frames at the server in order to adapt its video quality. During degraded quality operation, file accesses at the video server are strided non-sequentially.

In addition to resource-level variations, I/O latency variations can also be problematic for

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.

continuous media applications. While buffering can be used to hide variations in latency, it increases end-to-end latency which may become unacceptably high. Hence, predictable, low latency I/O is desirable.

The combination of complex access patterns with predictable, low-latency I/O requirements is what causes problems for current file systems. Non-sequential accesses are supported via the seek operation, which can be viewed as an imperative meta-interface for file I/O. In effect, it allows applications to override the system's default (sequential) file access behavior. However, as a consequence it also renders the system's prefetching mechanisms ineffective and makes device access latency visible to the application.

The goal of our research is to preserve the simple, sequential byte-stream model of file accesses, and to hide device access latency, even for applications with complex access patterns. The problem we have identified is that the seek interface to complex data accesses adds complexity to applications, and hides information from the system that could be used to improve performance.

This paper presents a solution to this problem, called synthetic files. Synthetic files can be viewed in several different ways. The data-oriented definition of synthetic files is that they are specialized views on regular files. They can be created dynamically, incrementally, or even optimistically, to match the access behavior of an application. The access-pattern oriented definition of synthetic files is that they are a declarative meta-interface for communicating future file access behavior to the file system. This information can then be used by sophisticated prefetching mechanisms to hide device access latency during the construction of an in-memory representation of a synthetic file. Hence, the creation of a synthetic file effectively converts a series of complex accesses to one or more underlying files into a set of sequential synthetic file accesses that can be satisfied from memory.

Synthetic files are described at creation time via programs written in a domain-specific language. They are subsequently accessed in sequential streaming mode. In contrast to the conventional approach of explicitly moving the file offset during reading, this two-stage approach has the advantages of providing access information far enough ahead of time to allow prefetching to occur, in addition to maintaining the simple streaming model of file I/O.

In this paper we describe an implementation of synthetic files in which both file prefetching and the incremental creation of synthetic files from prefetched file data are controlled via feedback. This use of feedback control allows prefetching to adapt dynamically to variations in device access latency, which might arise due to changes in load, or synthetic file accesses that span devices. It also allows the rate of synthetic file creation to adapt to match applications with variable I/O rates.

This paper is structured as follows: Section 2 presents the synthetic file interface. Section 3 describes implementation alternatives for synthetic files. Section 4 relates our experiences using synthetic files in adaptive multimedia applications. Section 5 presents our measurements of the performance of applications and the overheads of synthetic files. Section 6 discusses the advantages and tradeoffs of synthetic files, and Section 7 describes related work. Finally, Section 8 summarizes our experiences with synthetic files and describes future work.

2 Synthetic Files

Synthetic files present a sequential stream of data, read via the traditional file interface. The stream contents are determined dynamically by interpreting a synthetic file description provided to the file system via a meta interface.

At the lowest level, a synthetic file description is a list of data units that are concatenated into a single stream and read by the client application. While such a description is suffi-

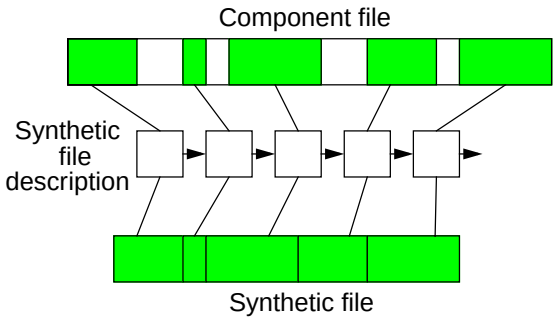

Figure 1: Synthetic file description. Synthetic files describe complex access patterns to component files. An application describes its accesses in domain-specific terms. This description is translated into an assembly of data units for lower layers of the system.

cient to allow file synthesis, in practice it is often cumbersome for clients to generate such a list. We utilize domain specific languages to translate application-level synthetic file descriptions into the lower-level data unit specifications. Figure 2 summarizes the synthetic file interface.

Synthetic files can thus be used to express the complex accesses of applications that stride through a file's data in arbitrary patterns. For multimedia applications, synthetic files can express variations in access patterns due to quality adaptation or playback changes, such as reverse or fast-forward. Database file accesses, such as indexed joins or query operations, could be expressed via synthetic files. Section 4 describes an implementation of a domain specific language that creates synthetic file descriptions from high-level quality of service adaptation policies.

`synthetic_file_open(dsl-descriptor)` - takes a domain-specific synthetic file description and returns a file descriptor to be used by read and synthetic_file_update.

`synthetic_file_read(fd, buffer, length)` - returns the next length bytes from file identified by fd into buffer.

`synthetic_file_update(fd, dsl-descriptor)` dynamically replaces the existing synthetic file description for fd. Subsequent reads will be satisfied via the new description.

Figure 2: Synthetic file API summary.

2.1 Consistency Issues

The semantics of the read operation on a synthetic file are easy to understand. It is less obvious what other file operations should do when applied to a synthetic file. We argue that seek should not be supported for synthetic files, and describe alternatives for handling writes to synthetic files.

The file seek operation is a meta interface to the read operation, that alters normally inaccessible implementation state (the file pointer) in order to enable complex access patterns. We present synthetic files as an alternative meta interface for complex accesses that obviates the seek operation. In cases when an access pattern is not known at synthetic file creation time, the synthetic file description can be updated dynamically. Not only is defining seek on synthetic files redundant, but it also interferes with the access pattern information that allows the underlying system to optimize itself. For these reasons, we have decided that the seek operation should not be defined for synthetic files.

There are two aspects of changes to a synthetic file. The first is defining a new view of the synthetic file. This type of change is accomplished by updating the synthetic file description via `synthetic_file_update`. The other aspect is handling writes a synthetic file or its component files. There are two directions of possible data flow that must be addressed: whether writes to synthetic files affect component files, and whether writes to component files affect synthetic files. The relationships between a synthetic file and its components could be bidirectional: writes to any component file appear in any synthetic file that includes it, and vice-versa.

An alternative to bidirectional consistency for writes is to prevent *any* changes to either synthetic files or component files to show up in the other. Such copy-on-write management significantly simplifies the data consistency aspect of writes, but introduces significant complexities of its own (e.g., the shadow chain management of Mach's virtual memory [19]).

An intermediate option is to handle writes to synthetic files by writing through to the underlying file buffer cache, to be handled by the filesystem's write-consistency policy. In the other direction, writes to component files appear in a synthetic file only if that portion of the synthetic file has not been realized in memory. This is the implementation choice we prefer. However, because all of the benefits of synthetic files can be explored without considering writes, our prototype does not support updates to either synthetic or component files.

2.2 Synthetic File Representations

A synthetic file is an abstract description of a file's contents. As such, it can be represented in many ways, with different benefits and tradeoffs.

The simplest representation of synthetic files is to fully realize them as a separate file on disk upon creation. This representation results in an eager copy of the data, and it makes non-write-through writes very simple to implement. However, the overall overheads could be impractical, since synthetic files are intended for large data sets, and multiple copies of gigabytes of data is infeasible. In addition, this representation eliminates the possibility of using synthetic files on streaming data. Finally, an eager realization of synthetic file data greatly reduces the efficiency of dynamic updates of synthetic file descriptions, which is required for QoS-adaptive applications. We address this in detail in Section 4.1.

At the opposite end of the spectrum is to delay realizing the synthetic file data until the data is actually requested by the application. This alternative results in the least possible space overhead (both on-disk and in memory), but ignores the possibility of using synthetic file descriptions as access pattern hints to optimize data prefetching.

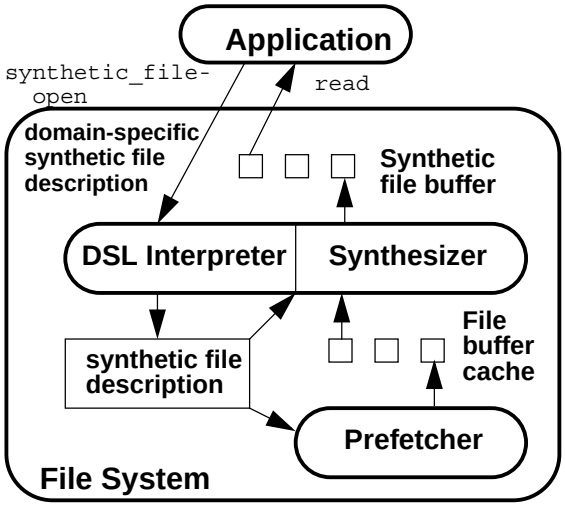

Figure 3: The interactions between an application, the file synthesizer and the prefetcher. The synthesizer interprets the domain-specific synthetic file description to determine the data to fetch, and translates it into the description needed by the prefetcher.

Our choice for representing synthetic files is to incrementally interpret the synthetic file description, and to realize the data in memory ahead of time, according to a feedback-controlled threshold. Since synthetic file components need not be page or block aligned, but file reads must be, data is realized by reading from pages of the file buffer cache. We drive the filesystem's prefetching into the buffer cache by feedback control as well, according to observed latencies to the underlying devices. We describe these layers and the feedback control in Section 4.

3 Implementing Synthetic Files

Figure 3 depicts the components and interactions in our prototype implementation of synthetic files. The application creates a synthetic file by specifying its contents in a domain-spe-

cific language. The synthesizer constructs the synthetic data stream by interpreting the domain-specific synthetic file description and producing the data-unit list of synthetic file components. It shares this domain independent synthetic file description with the prefetcher. The synthesizer uses the description to read the requested data units from the underlying file system. The prefetcher uses the description to determine the data to prefetch.

The separation of duties in this design simplifies the construction of the feedback systems used to control the data-fetching workahead. The synthesizer's feedback-driven controller adapts to variations in the rate of application reads and thus masks high-frequency application variability from the underlying prefetcher.

The task of the prefetching layer is to adapt to variations in device latency and in the average access rate of the synthesizer. Just as the synthesizer masks application variability from the prefetcher, the prefetcher provides device independence to the synthesizer. We present details of the feedback systems of both layers in Section 4.2.

4 Synthetic Files for Adaptive Multimedia

This section describes our experience using synthetic files in the context of a quality of service-adaptive distributed multimedia application.

4.1 Quality of Service Adaptation

Streaming media formats can be encoded with multiple layers of quality information. When resources become scarce, the system can adapt by discarding information from different layers, resulting in a reduced quality lower bandwidth stream. Synthetic files enable a clean separation between the concerns of playing a media stream and the concerns of adapting the stream to available resources.

We have created a synthetic file description language that corresponds to quality adaptation via data dropping. The data format is a modification of MPEG-1 that encodes a video stream with two axes of adaptation: spatial resolution, and frame rate (temporal resolution) [18]. As available resources vary, the stream can be adapted by dropping packets that correspond to either dropping frames or reducing spatial resolution. The policy for choosing which axes to adapt along can be succinctly encoded by assigning a priority to each packet according to its relationship to preferred quality axes. Adaptation in this system consists of varying a priority threshold, below which packets are discarded. Thus the synthetic file description consists simply of specifying the current dropping threshold. The synthesizer translates this specification into the lower-level file access patterns.

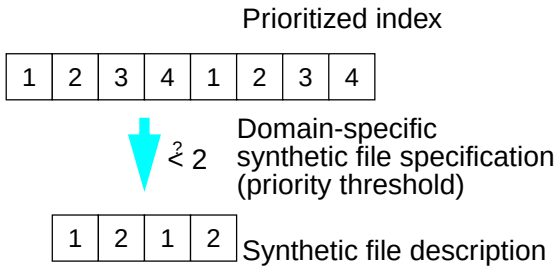

Figure 4: Domain-specific synthetic file specification for adaptive video streaming. The adaptive media stream separates quality axes into separate data units. The adaptation policy labels each unit with its component's priority. The synthetic file can be expressed with a simple priority threshold, which is interpreted by the synthesizer to create a synthetic file

Synthetic file specifications for our data-dropping virtual machine are described by specifying three parameters: a source file, an index, and a dropping threshold. This model assumes that all the constituent data is contained in a single file, but it can be extended to support more general cases. Figure 4 depicts the construction of a synthetic file description given an index and a priority threshold. The index is a list of data units, each unit named by

offset and length in the source file, and prioritized by the QoS adaptation policy. Without the interposition of the synthetic file abstraction an application using the traditional file interface would have to interpret the index and issue a stream of seek and read requests in order to access the desired stream of data. As available resources vary, client applications may 're-specialize' the contents of a synthetic file by specifying a new dropping threshold. The new threshold is applied to all data read after its application.

4.2 Feedback-Control of Prefetching and Synthesis

Many file systems optimize caching and prefetching policies for simple sequential accesses [5, 6, 11]. Synthetic files make these optimizations available to more complex data access patterns, but they also complicate the task of prefetching, since block-to-block access times can vary significantly as a synthetic file references data with long and/or variable strides. Complicating matters further, since synthetic files are designed for QoS-adaptive applications, the system has to adapt to variations from the application as well.

Our implementation of synthetic files uses software feedback [3, 10] to dynamically adjust the prefetch horizon based on observed and predicted device latencies.

We used the SWiFT toolkit [3] to construct feedback-based rate controllers for both synthesis and prefetching. The synthesizer produces data for the application and consumes data from the prefetcher. In turn, the prefetcher produces data for the synthesizer and consumes data from I/O devices. The rate controller at each level monitors the rate at which data is being read by its consumer, calculates a target fill level and adjusts the rate it reads data from the producer. The equation used to calculate the target fill level is based on the following equation that the TIP system [13]. uses to statically determine prefetch depth:

*Buffer_Size = Access_Latency * Access_Rate*

In contrast to TIP we dynamically estimate *Access_Latency* and *Access_Rate* by using system monitoring. This formula adjusts *Buffer_Size* in response to changing conditions in the system.

5 Experimental Results

This section describes how we tested our prototype synthetic file server. We used the software oscilloscope provided by the SWiFT toolkit to observe monitored and controlled variables in our system.

The experiments demonstrate the operation of our feedback circuits in reaction to I/O latency changes and to application adaptation. The hardware we ran the experiments on is a 300MHz Pentium II, with 128 Megabytes of DRAM and a 4.3 gigabyte Quantum Fireball SCSI disk. Our prototype synthetic file implementation is a stand-alone user-level server running on top of a Linux 2.0.30 kernel.

In order to isolate the I/O behavior of the system, our application simulates the read behavior of an MPEG player, but does not incur the CPU overheads of decoding or display. In the experiments below, the player is simulating varying degrees of quality adaptation, which corresponds to complex, strided access patterns.

Figure 5 captures the system reactions to the increased I/O latency incurred by introducing competing load over a period of four seconds. The top oscilloscope shows the measured latency of disk reads, the calculated mean deviation of disk reads, and the number of outstanding requests to the disk. Disk latency immediately increases when competing load is introduced, due to competition for the disk head and I/O queue. The deviation in I/O latency also increases with the contention, which the prefetcher takes into account.

The middle oscilloscope in Figure 5 shows the prefetch depth and the prefetch target over the same period. One thing to notice is that the

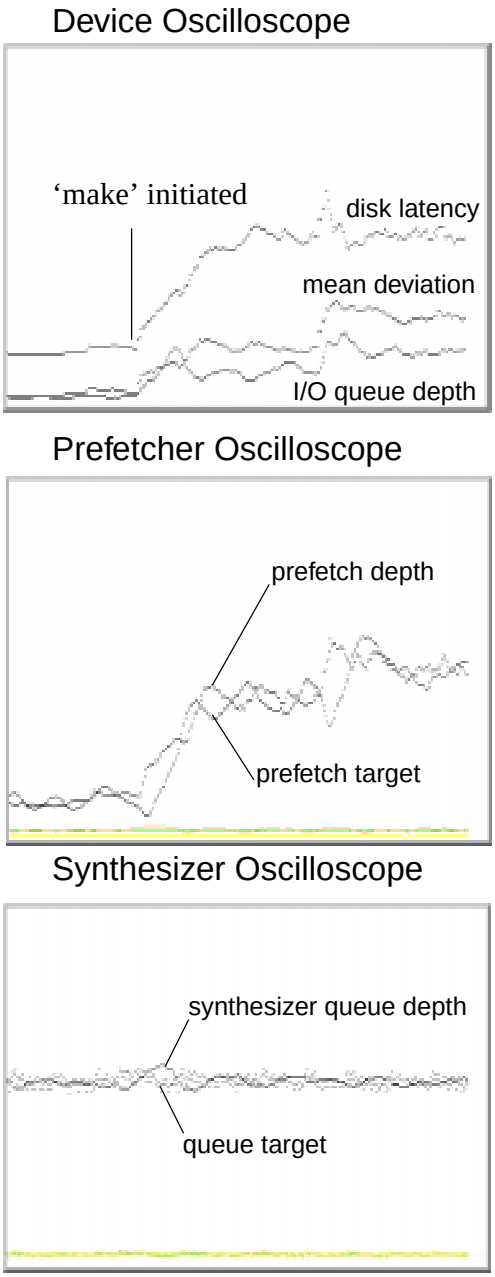

Figure 5: Adapting to I/O contention. These oscilloscope snapshots depict the I/O status of various levels of the system. A 'make' was initiated at the point indicated in the top graph, which introduced contention to the disk, resulting in increased I/O latencies. As expected, the prefetcher adapts to the load, hiding it from the synthesizer.

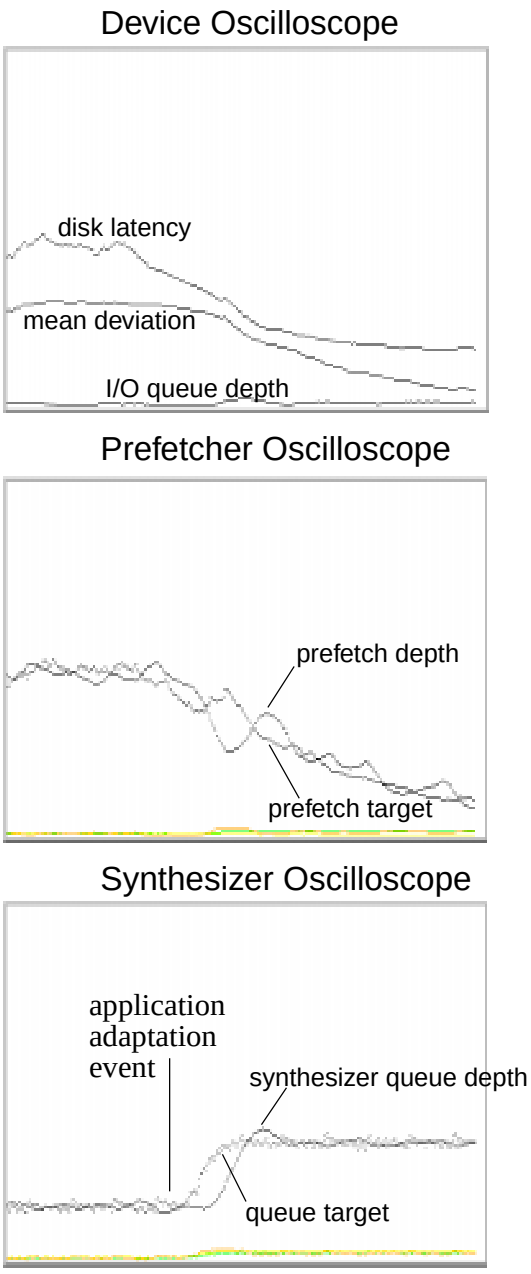

Figure 6: Adapting to application-level adaptation. This set of oscilloscope snapshots show the system reacting to an application adaptation. The application has adapted to improve quality. The bottom chart shows the synthesizer reacting to the application by increasing its queue depth to match the application's higher rate. The improved sequentiality of the higher data rate stream results in *lower* I/O latencies and correspoinding lower prefetch depth.

prefetch depth momentarily drops while the controller adapts the prefetch rate, but that the prefetch depth was sufficient to prevent underflow.

Finally, the bottom oscilloscope shows that the synthesizer – and hence the application – is unaffected by the change in load.

Figure 6 demonstrates the system reacting to the application dynamically adapting the synthetic file by raising the priority threshold in order to improve quality. In contrast to the previous experiment, the event flow is driven by the application instead of the disk.

The bottom oscilloscope shows the synthesizer reacting to the application adaptation by increasing its target queue depth (corresponding to the increased access rate).

The top oscilloscope shows the disk latency improving in response to the adaptation, which may be counterintuitive. The data stream resulting from the increased priority threshold has a shorter stride, which improves sequentiality, and corresponding disk performance. The middle oscilloscope shows the prefetcher adapting to the improved disk latency by reducing its prefetch depth.

To summarize our experimental results, we have shown that the feedback-based rate controllers in the prefetcher and synthesizer are able to adapt to changes in both system and application behavior.

6 Discussion and Future Work

Files are used to store and retrieve data with different degrees of structure, from raw ascii to HTML to highly structured data such as MPEG files and databases. It is testament to the flexibility of traditional files that they can accommodate so many different types of data. One way to view synthetic files is that they present a way of integrating some of the benefits of previous structured filesystem interfaces with the untyped byte-stream file interface. Synthetic files do not compromise the flexibility of untyped file interfaces since the structure is provided by application programmers via domain-specific synthetic file descriptions. Providing this structure to the file system enables it to be used to improve resource management decisions, thus improving performance.

A synthetic file description is a *specialized* view of a generic untyped data file. As with specialization of code [14] synthetic files can be created statically, dynamically, incrementally, and even optimistically. Specializing data can result in additional invariants that can be used to enable code specialization. For example, since synthetic files can be used to guarantee sequential reads, even for complex data, this guarantee enables code specialization to generate optimized read calls to access them efficiently.

The relationship between code and data specialization extends to the underlying implementation of synthetic files as well. Given the synthetic file description, a specialized read call could be generated that implements the underlying complex data accesses and seeks. Alternatively, as with synthetic files, the application can explicitly disclose the structure of the data, and allow the file system to specialize the data, which it reads via an unspecialized interface, as measured in the experiments of Section 5. Finally, we plan to explore combinations of the two approaches, to drive system specialization via information extracted from synthetic file descriptions.

7 Related Work

In our earlier work on adaptive prefetching [15] we proposed a mechanism for hiding device access latency. Synthetic files extend this mechanism in three ways. First they simplify the application interface by implementing a sequential byte-streaming model for reading data. Second, their incremental construction under feedback control offers better support for applications with variable-rate I/O. Third, they constitute a *meta-interface* [7] for communicating access pattern information to the prefetcher.

The use of access pattern information to drive prefetching is not new. Kotz investigated automatic detection and prediction of complex access patterns [8]. Because of the limits of automatic prediction-based approaches, more recent filesystem research has proposed *informed* interfaces through which applications disclose information about upcoming access patterns. TIP-2 [13], for example, uses "disclosure hints" of upcoming accesses to drive its buffer management policy. Clients of TIP-2 must explicitly synchronize their data accesses with the disclosures they provide to the meta-interface. In contrast, synthetic file clients communicate their access pattern only once, in the specification of a synthetic file. Our implementation uses the same prefetch depth equations as TIP-2, but adapts the estimations dynamically, whereas TIP-2 does it statically.

The SETS system [16] provides an informed interface that does not require applications to synchronize hints with accesses. An application describes a "dynamic set" of files that it wants to access. The system uses this information to manage prefetching and buffering, and is free to define an order on file accesses to improve efficiency. This reordering behavior is not appropriate for the I/O requirements of continuous media applications, however. Furthermore, SETS is whole-file based, while synthetic files accommodate complex accesses within, as well as among, files. Finally, neither SETS nor TIP-2 adapt dynamically to variable-rate applications or to variable device access latencies.

The meta-interface provided by synthetic files is similar to the layer of indirection provided by the Mach external pager interface [19]. In fact, one possible implementation of synthetic files would be to replace the default external pager for Mach files, called the inode pager, with a pager that accepts and interprets synthetic file description programs. This implementation would be preferable in some ways to the technique we chose, due to more efficient communication between the layers of buffer management. However, taking advantage of this facility sacrifices portability, since the external pager facility is not available even on many systems derived from Mach (e.g., Digital Unix and the NeXT OS).

Another view of the synthetic file interface is that it provides application-defined structure to data at the file system level. This feature is reminiscent of early structured file systems, such as those described in [4]. In these systems applications defined file data formats, which were stored on disk in variable-length records, and could be indexed and accessed with different patterns. The synthetic file facility is more powerful and flexible than these systems: its synthetic file descriptions are more abstract, being specified in domain-specific terms; synthetic file contents can be generated dynamically, "just in time" to satisfy access requests; and it preserves the efficiency of an underlying block-oriented file implementation.

Previous work in continuous media and real-time file systems is also related to synthetic files. Continuous media file system research has largely focused on optimizing overall throughput and guaranteeing individual stream bandwidth to simple, non-adaptive media clients [1, 2, 9]. Real-time file systems leverage the real-time facilities of the underlying operating system, and combine careful data layout and disk scheduling to provide latency guarantees for file system requests [12, 17]. In contrast, synthetic files aim to support *QoS-adaptive* clients instead of offering bandwidth guarantees, and offer low-latency I/O for synthetic file accesses, even when they span devices with greatly different performance characteristics.

8 Conclusions

Files are an effective interface to streaming data, but their predictability and performance is impeded when applications use the seek meta-interface in order to implement complex data access patterns. We have presented syn-

thetic files, an alternative meta-interface to complex data accesses, which enable the separation of data access specification and the data accesses themselves. This separation simplifies applications, and allows the data access specification to be passed down through system layers, in order to enable sophisticated prefetching and buffering optimizations.

Our implementation of synthetic files uses two separate feedback control circuits. The first feedback system controls the amount of work-ahead for realizing the synthetic file in memory, which adapts to variations in application behavior. The second feedback system controls the prefetch depth in the filesystem buffer cache in order to mask variations in device latency.

We demonstrated the effectiveness of synthetic files by using them to implement the complex data accesses of a QoS-adaptive video player. The experiments verified that the prefetching layer effectively hides variable device latencies from the application, and that the synthesizer adapts to variable application behavior. We demonstrated that the system as a whole is able to accommodate complex accesses to variable-latency devices from dynamically-adaptive applications. Finally, all of these benefits are provided with a more natural application interface than the existing read/seek calls provided by existing systems.

9 References

[1] Anderson, D.P., Y. Osawa, and R. Govindan, *A File System for Continuous Media.* ACM Transactions on Computer Systems, 1992. **10**(4): p. 311–337.

[2] Bolosky, W.J., R.P. Fitzgerald, and J.R. Douceur. *Distributed Schedule Management in the Tiger Video Fileserver.* in *Sixteenth ACM Symposium on Operating Systems Principles.* 1997. Saint-Malo, France.

[3] Cen, S., *et al. Demostrating the Effect of Software Feedback on a Distributed Real-Time MPEG Video Audio Player.* in *Demonstration at the 1995 ACM Multimedia Conference*. 1995. San Francisco, CA.

[4] Crowley, C., *Operating Systems: A Design-Oriented Approach*. 1997, Chicago: Irwin.

[5] Custer, H., *Inside Windows NT.* 1993: Microsoft Press.

[6] Digital Equipment Incorporated, *Digital Unix*, 1995: Waltham, MA.

[7] Kiczales, G. *Towards a New Model of Abstraction in the Engineering of Software*. in *Proceedings of IMSA 1992 Workshop on Reflection and Meta-level Architectures*. 1992.

[8] Kotz, D.F., *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors*, in *Department of Computer Science*. 1991, Duke University: Durham, North Carolina.

[9] Lougher, P. and D. Shepherd, *The Design of a Storage Server for Continuous Media.* The Computer Journal, 1993. **36**(1): p. 32-42.

[10] Massalin, H. and C. Pu, *Fine-Grain Adaptive Scheduling Using Feedback.* Computing Systems, 1990. **3**(1): p. 139-173.

[11] McKusick, M.K., *et al.*, *A Fast File System for UNIX*. Transactions on Computer Systems, 1984. **2**(3): p. 181-197.

[12] Molano, A., K. Juvva, and R. Rajkumar. *Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach.* in *IEEE Real-Time Systems Symposium*. 1997.

[13] Patterson, R.H., *et al. Informed Prefetching and Caching*. in *Fifteenth ACM Symposium on Operating Systems Principles*. 1995. Copper Mountain Resort, Colorado.

[14] Pu, C., *et al. Optimistic Incremental Specialization: Streamlining a Commercial Operating System*. in *Symposium on Operating Systems Principles (SOSP)*. 1995. Copper Mountain, Colorado.

[15] Revel, D., *et al. Adaptive Prefetching for Device Independent File I/O*. in *Multimedia Computing and Networking*. 1998.

[16] Steere, D.C. *Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency.* in *Sixteenth ACM Symposium on Operating System Principles.* 1997. Saint-Malo, France.

[17] Tezuka, H. and T. Nakajima. *Simple Continuous Media Storage Server on Real-Time Mach*. in *USENIX 1996 Annual Technical Conference*. 1996. San Diego, CA.

[18] Walpole, J., *et al.*, *Quality of Service Semantics for Multimedia Database Systems*, 1998, Oregon Graduate Institute.

[19] Young, M.W., *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*, in *School of Computer Science*. 1989, Carnegie Mellon University: Pittsburgh, PA.