Constrained-Latency Storage Access: A Survey of Application Requirements and Storage System Design Approaches

Richard Staehli

Jonathan Walpole

Oregon Graduate Institute
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999
{staehli ,walpole}@cse.ogi.edu

October 10, 1991

Abstract

Applications with Constrained Latency Storage Access (CLSA) are those that have large storage needs and hard constraints on the amount of latency they can tolerate. Such applications present a problem when the storage technology that is cost effective and large enough cannot meet their latency constraints for demand fetching. Examples are found in the developing field of multimedia computing and, to a lesser extent, in real-time database literature.

This paper examines the nature of timing constraints at the application-storage interface and defines a classification for both the synchronization constraints of the application and the latency characteristics of the storage system. This classification is then used to survey existing approaches to CLSA and to assess their limitations. The more promising technologies are identified, and their suitability for integration into a general purpose storage management system to meet CLSA needs is examined.

Keywords: Real-Time Storage Systems, Multimedia, Operating Systems

1. Introduction

Storage systems have typically been designed to provide optimal average performance without requiring explicit knowledge of their client application's data access behavior. The widespread introduction of multimedia computing gives storage system designers reason to change this approach. Digital audio and video, referred to as continuous media, are expected to be incorporated as common data types in many new multimedia systems [Fren89], [Phil91],

[Litt90a], [Luth89]. These media types and the applications that contain them require both large amounts of storage and real-time access [Litt90a]. In order to store the large amounts of data needed for video, a storage system must use archival storage devices, such as optical disks, which are less expensive, but slower than magnetic disks. This approach has been taken for both personal computer based applications [Luth89] and in larger experimental multimedia architectures [Chri86], [Berr90]. These systems are able to read the next frame in the sequence from the optical disk at near maximum bandwidth but they have no mechanism for anticipating a seek to a new location on the same or a different disk. The slow seek times of optical disk drives present a real challenge to applications that must display a new video sequence in synchrony with another video or audio stream that is already running. Multimedia applications will frequently require that separate data streams such as audio and video tracks be synchronized [Litt90b], [Hodg89].

An application as simple as a slide-tape presentation illustrates how some applications must explicitly anticipate storage latencies. A digital slide-tape presentation may require that slide image *n* from a database be displayed within 0.1 second of synchronization audio event *n* in a sound track, so that the viewer perceives the two events as simultaneous. Once the presentation of the sound track has begun it cannot be paused. Suppose that each stored image consumes 1 MB, the presentation contains 100 images and the synchronization events occur 1-10 seconds apart. The platform consists of a workstation with 16 MB of memory and the database includes thousands of images on several platters of an optical disk juke box. A realistic seek time for current optical disk technology is 0.5 second and disk replacement in a juke box may take 4 seconds. Clearly it is not feasible to wait for a synchronization event before attempting to retrieve the corresponding image.

This paper examines a class of applications that require read access to a larger storage capacity than main memory and yet cannot tolerate the worst-case latency of secondary storage access.[1] The term *constrained latency storage access* (CLSA) is used for applications that have hard deadlines for the completion of some storage accesses. A hard deadline for a task indicates that the value of completing the task drops sharply after the deadline is past [Abbo88]. Examples of such applications are found not only in multimedia applications with continuous media but also in real-time databases, which must look to prefetching in order to satisfy strict constraints on transaction times [Krat90]. Many applications would benefit by lower latency storage access but only a restricted set have hard deadlines.

In CLSA applications, quantity of data, cost factors and physical distribution necessitate storage architectures with *large*[2] worst-case latencies. To satisfy constraints, worst-case latencies must be avoided by exploiting knowledge of the storage system and the CLSA application's data access pattern. Storage hierarchies are a well known technique for providing both low average storage cost and low average access latency [Cohe89]. All data is logically stored at the

---

[1] Temporal constraints on the completion of write access are less common and are beyond the scope of this survey.

[2] The term *large* is used in a relative sense. Latency is considered to be *large* if it exceeds an application's access time constraints.

lowest level of the hierarchy where the cost per byte is least. Each higher level is *faster* (i.e. has a lower latency) and is used to cache frequently accessed data from the level below. In the simplest cases, a hierarchy will consist of magnetic disk storage and main memory buffers, as in the Unix file system. If a CLSA application is to execute correctly with a storage hierarchy, the upper levels of the hierarchy must be *fast enough* to satisfy all latency constraints. To ensure that data is available in fast storage to meet CLSA needs, storage access can be defined in terms of a *fetch-demand* model. The *fetch event*, which initiates the copying of data up into fast storage, must precede the application's *demand* for the data by an amount that depends on the anticipated latency of the copy process. The definition of a *fetch event* is naturally extended to include the initiation of actions which do not involve copying,but that never-the-less promote some data to a state where access latency is reduced. For example, seeking to a new track of a magnetic disk reduces the subsequent access times for sectors in that track.

The remainder of this paper classifies CLSA applications and surveys a range of storage system design approaches. Section 2 presents a more precise characterization of application requirements and the causes of storage system latency. Section 3 compares existing approaches to simple versions of the CLSA problem and identifies their strengths and limitations. A general approach to the CLSA problem is outlined briefly in Section 4, followed by discussion in sections 5, 6 and 7 of three aspects of the approach. Sections 5 and 6 discuss real-time scheduling technologies and resource reservation protocols, and Section 7 surveys scripting languages. Section 8 concludes the paper.

2. Characteristics of Constrained Latency Storage Access

Before surveying current approaches to CLSA system design it is useful to define characteristics for classifying applications and storage systems. The types of constraints that arise in CLSA systems are defined and related in Section 2.1. These definitions are then used in Sections 2.2 and 2.3 to present a simple classification scheme for applications and storage systems respectively. With these classification tools, approaches to system design are more easily identified that can satisfy the broader range of CLSA applications and storage architectures.

2.1. Definition of constraints.

By definition, a *value function* for an event says that if that event occurs at time *t* relative to a reference event *r*, the event has value *V(t)* [Abbo88]. A value function is the most general specification of a temporal constraint in that it gives a partial ordering for the domain of possible event times. A constraint is *satisfied* if the value of the event at the time it occurred was above an acceptable threshold. Figure 1 gives some examples of value functions. Figure 2(a) indicates a *soft* constraint that the event should occur at about the same time as the reference event. Figure 2(b) shows a *hard* constraint that the event occur within some short period after the reference event. A hard constraint is frequently

represented as a step function that rises sharply at some point, then drops again to zero or a negative value after reaching a hard deadline.

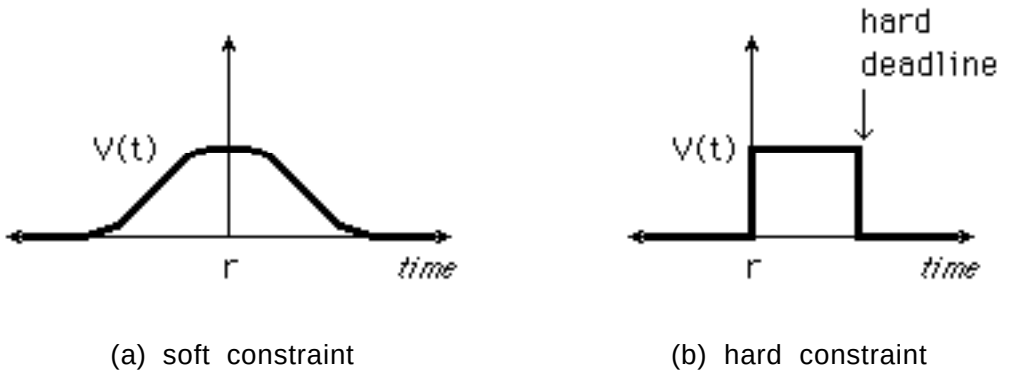


Figure 1. Value functions relative to the occurrence of an event r.

Two constraint relations are of major importance for CLSA applications: the application's synchronization constraints and the storage system's latency constraints. These constraints can be characterized in terms of three events within a storage system: *demand*, *fetch* and *ready* events. A *demand* event occurs when the application requests data from storage. The retrieval of data from storage begins with a *fetch* event and completes with a *ready* event when the data is available to the application. As illustrated in Figure 2, non-real-time storage uses typically consist of demand-fetching where a fetch is triggered directly by a demand.

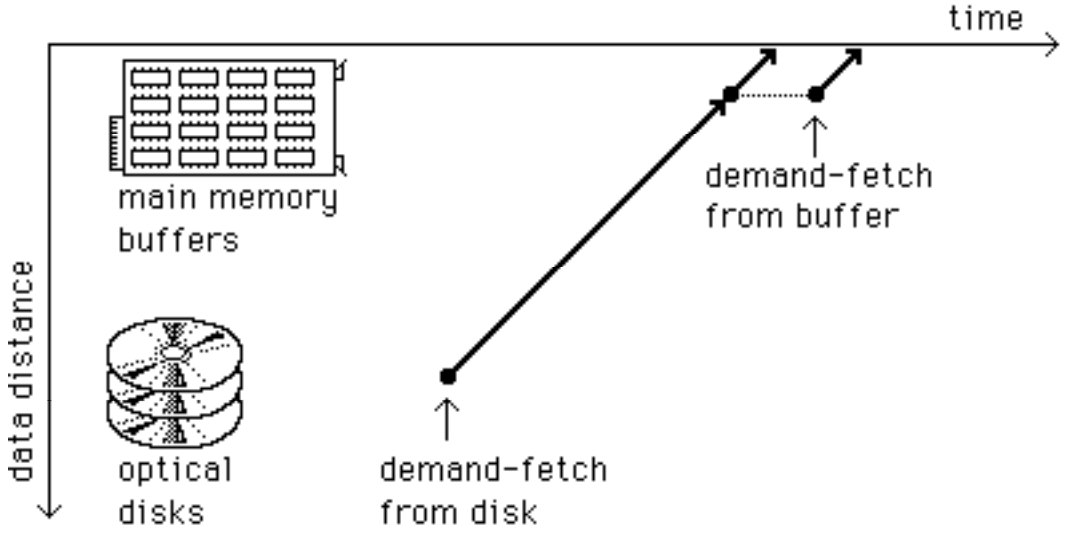


Figure 2. Demand fetching. (Data distance measured in access time.)

CLSA applications, by definition, must allow the fetch to precede the demand. Prefetching implies that the storage system can bring data closer (in access time) to the application and hold it there until the application demands it. An application's latency constraint defines a logical level of storage, called the *constraint threshold*, above which the constraint can be satisfied by demand-fetching. The constraint threshold is illustrated in Figure 3.

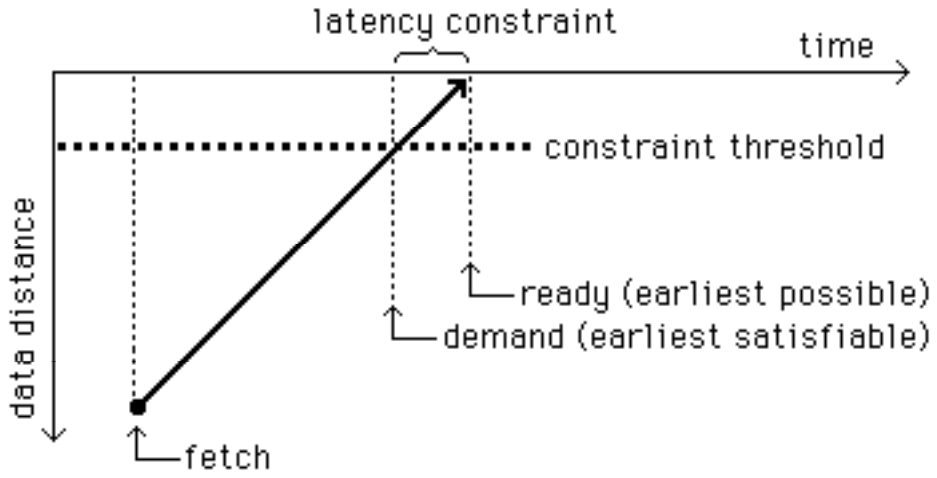
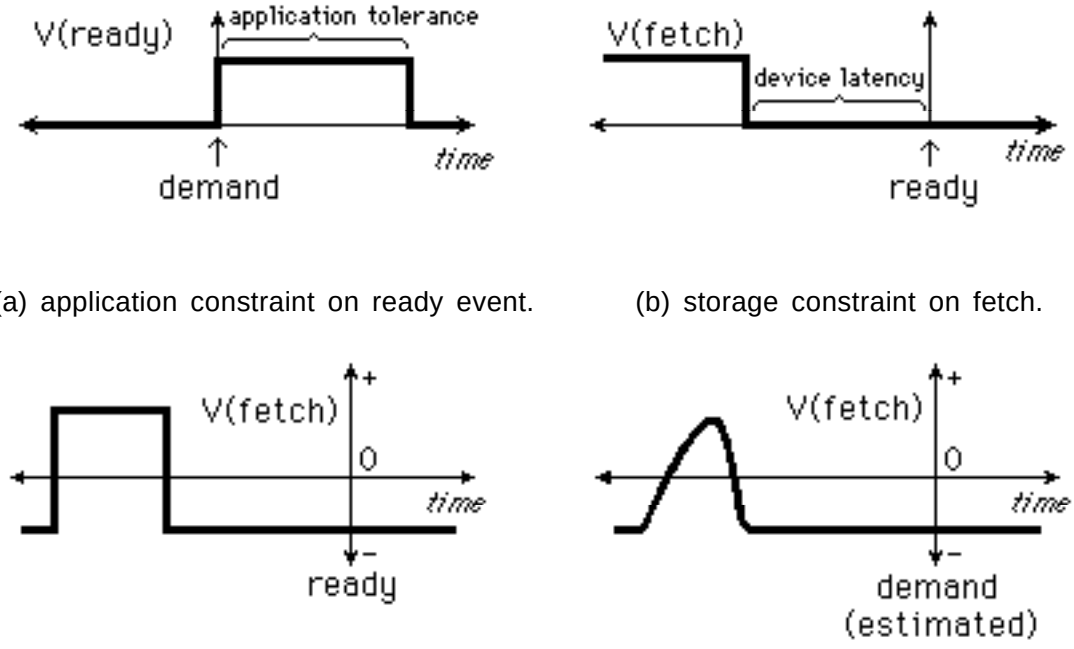


Figure 3. Defining the constraint threshold level of storage.

An application may require that data be ready within some short period following a demand event as illustrated in Figure 4(a). The latency characteristics of a particular storage device will also constrain a fetch to precede the ready event as illustrated in Figure 4(b). The pair of constraints on the same ready event serves to indirectly relate the fetch to the application's demand event, i.e. to satisfy both constraints the following equation must hold:

$$t_{fetch} + \text{latency} < t_{demand} + \text{tolerance}$$

By definition, all data for a CLSA application cannot be held simultaneously in fast access storage (i.e. logical levels above the constraint threshold). In fact, fast storage space is scarce and will fill up unless some data can be removed. The availability of fast storage is a function of it's size, the value of other data in it and the cost of running the replacement algorithm. Until some data in fast storage has less value than the next data to fetch, the value of fetching will be zero or negative. The value function in Figure 4(c) reflects a restricted availability of fast storage. These value functions are idealized in that they assume perfect knowledge of application requirements and storage system resources. If only a probability distribution is known for when space will be available in the buffer and when the application demand will occur then a softer curve will result as shown in 4(d). A *feasible schedule* is one in which all fetch and ready events are scheduled in a way that meets both the application requirements and storage system capabilities.



(a) application constraint on ready event.

(b) storage constraint on fetch.

(c) addition of resource constraints.

(d) probabalistic knowledge.

Figure 4. Qualitative definition of value functions for storage events.

2.2. Predictability of demand events.

Section 3 classifies the approaches to CLSA system design according to the complexity of the applications that they can support. Timing of storage demands as well as the pattern of storage locations accessed both contribute to an application's complexity. The complexity of demand timing can be measured by the amount of information needed to accurately predict demand event times. The following list divides all timing information into three categories, from simple periodic, to uncertain knowledge which can only be expressed probabalistically:

- Periodic events occurring at regular intervals.
- Scheduled sporadic events that can be anticipated well in advance.
- Probabalistically predicted events.

More than one set of timing requirements may need to be considered in a given application. For example, a video presentation may specify limits on both skew and jitter [Litt90a]. Skew is the average amount that the presentation of video frames deviates from a periodic schedule. Jitter is the amount of variation in the time interval between frames. The skew limit gives rise to a constraint between demand and periodic events. The jitter limit, on the other hand, involves not a periodic or scheduled event but the actual presentation time of the previous frame, which is dynamically determined and thus uncertain.

In addition to the predictability of when demand events occur, application complexity can be measured by how much information is needed to predict access locations. A simple application may be known to access data sequentially, while more complex access patterns may require scripts or uncertain calculations:

- Sequential locations.
- Scripted locations that are known but may have no locality.
- Probabalistic predictions only.

Considering the example of video again, scripted presentations can include both sequential access patterns and jumps to new video segments. Some interactive video browsing and editing systems will also produce access patterns that can only be described probabalistically.

2.3. Predictability of storage latency.

Storage latency is a function not only of physical device characteristics, but also the policies that dictate the placement of data and when it is moved. Common device characteristics to be considered include:

- memory access speed
- bus contention delays
- disk controller overhead, seek, rotational delay and transfer rate
- mounting time for offline disks in a jukebox
- network communication delays

The policies of the storage system are evident in:

- data movement and replacement policy
- the handling of resource (including cpu) contention in multitasking environments
- decompression and other processing requirements

The delay that the application experiences between demand and ready events will be smaller than the latency between fetch and ready events when the fetch precedes the demand. In the Unix file system, a prefetching facility may read a block into the file buffer, incurring disk access latency, while a subsequent application demand for the same block experiences only the buffer access time. In general, the system designer may need to recognize many levels of storage and know the latency for fetches between each of them. Predicting latency is made more complex when the storage architecture includes many levels of storage and may be unpredictable when storage is shared with other systems. The simpler approaches surveyed in Section 3 satisfy a subset of CLSA applications by making restrictive assumptions about the storage architecture. Other approaches show promise for supporting general purpose storage architectures. The various approaches can be classified according to how much they restrict the storage architecture:

- Simple dedicated (single-use) storage with acceptable worst-case sequential access latency.
- Predictable, but possibly wide variability in latency as in storage hierarchies.
- Shared resources: latencies subject to contention among multiple users.

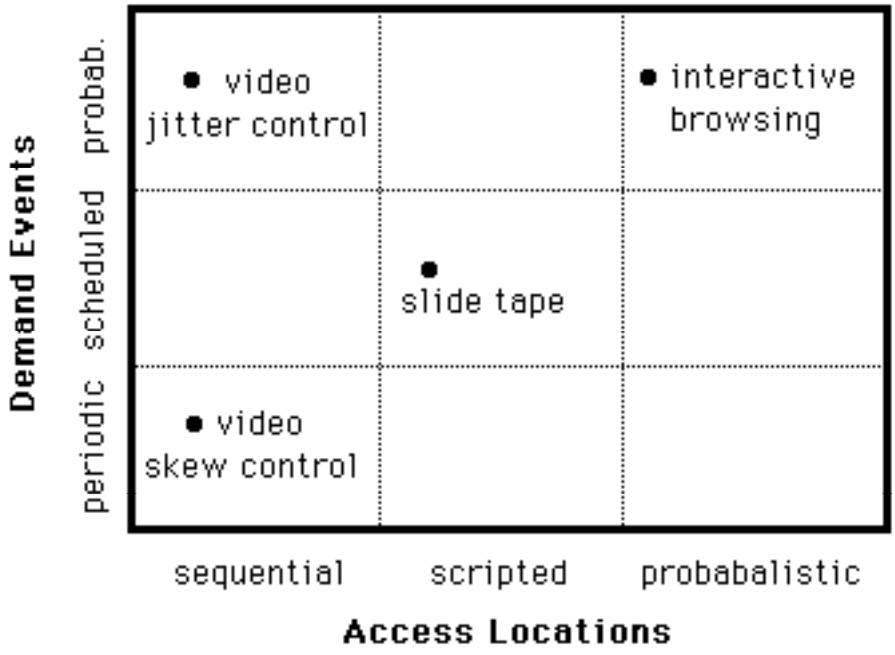


Figure 5. Predictability of storage demands.

3. Comparison of Current Architectures

This section surveys a variety of existing CLSA systems grouped by the similarities in their approach to the problem. The significant contributions are identified from each system and the limitations of the approach are discussed.

3.1. Non-solutions

These first two groups do not represent solutions, but represent common approaches that are used where hard latency constraints do not exist or are not a severe problem. In the latter case, when constraint violations are few or tolerably minor, the problem can be ignored.

3.1.1. Ignoring the problem.

Berra, et. al. present a general schema for representing temporal relationships between database objects in a multimedia presentation [Berr90]. Their schema uses Object Composition Petri Nets (OCPN) [Litt90b] which allow arbitrarily complex large grain synchronization relationships to be specified between the presentation of objects. To execute such a schema, they propose an elegant and simple algorithm that forks a new process for each concurrent task and inserts waits to accomplish delays between object presentation times. No prefetching is supported. Instead, objects are retrieved from the database as their presentation time arrives, with no allowance for unpredictable access delays. A subsequent paper [Litt90a], discussed below, addresses the fine grain synchronization requirements for continuous stream data.

3.1.2. Minimization.

Minimizing average latency has been pursued in virtually all operating systems and database architectures that include caching/buffering schemes. The techniques can be divided into two groups: replacement policies that attempt to retain the data that will be used again, and prefetching techniques that attempt to guess which data will be used in the near future. Replacement policies alone are clearly insufficient for CLSA problems as the first access to each stored object will never gain the benefit of low cache latency and a second access may not occur soon. Prefetching strategies can be effective if they guess correctly often enough to satisfy an application's tolerance for constraint violations.

Prefetching, as described by Wedekind [Wede86], is a technique to construct a dynamic, transaction-oriented sub-database in main memory. Before executing a “canned transaction” a second transaction is derived from the first and executed in order to prefetch and fix in memory, all the data needed. Data is fetched in the order that it is expected to be used and this occurs simultaneously with the primary transaction. The second transaction fetches data according to a superset query derived from the original. While this approach [Krat90] shows promise for large performance increases, it uses no knowledge of when the data is needed and so the primary transaction may still block while waiting for data. Furthermore, other concurrent transactions will compete for storage access with unpredictable consequences for actual latency.

Another sophisticated approach to prefetching is described by Palmer and Zdonik [Palm91]. The Fido predictive cache learns to recognize access patterns over time within specific application contexts. An associative memory is then used to recognize similar patterns and predict future access sequences. It prefetches data only when the application blocks on a fault and a familiar pattern is recognized from a sample of recent accesses. Prediction sequence lengths are limited by an arbitrary training parameter. The size of the sample can be varied in order to trade off the guess rate (which can lead to an error glut) against guess accuracy. A novel cache replacement policy is described that retains data that is predicted to be accessed soon as well as recently used data. Unused prefetches are replaced first. Fault reductions on the order of 50% under favorable conditions are reported in initial testing.

The major attraction of the above approaches is that the application does not have to be modified to provide fore-knowledge of storage accesses to the storage system. Applications in which timing information is unavailable may be able to benefit from increased performance when latency minimization techniques are used.

One limitation of these prefetching strategies is that they do not know *when* data will be needed, consequently they must fetch as rapidly as possible. If the prefetching does not begin soon enough, the consumer application may still have to wait for unavailable data. If they fetch too soon, data that was of more value may be replaced in the cache. Also, there is no information about the order in which to prefetch data for concurrent processes. Inaccurate fetch predictions have the potential to degrade performance by filling up the cache with unused data and wasting cpu and I/O bandwidth. Since no scheduling analysis is done, minimization approaches offer no way to know in advance if a CLSA application can execute correctly.

Scripted multimedia presentations typically have complete knowledge of when and what data is to be accessed. There is a corresponding lack of tolerance in such applications for delays and/or lost data. Little and Ghafoor [Litt90a] give an example of requirements for video telephony with maximum inter-frame jitter of 10ms. None of the minimization approaches above can meet such strict requirements for prefetch timeliness.

3.2. Calculated Optimizations

For home consumer products where cost reduction is a primary concern and for experimental platforms where the applications frequently tax the storage capabilities, it is natural to exploit as much knowledge of the system as possible to meet requirements, even at the expense of generality. The following techniques make specific assumptions about the access pattern and the computing platform, which limit their applicability to other problems.

3.2.1. Presequencing data in storage.

One source of variability in storage latency is a discontinuity in the sequence of storage locations. Seeking to another track on a disk, for example, is much more expensive that accessing the next block within the same or an adjacent track. Discontinuities can be eliminated by storing the data sequentially before it is retrieved. Presequencing is a technique that reduces the complexity of the access location sequence from scripted to sequential and generally gives a simple bound on access latency.

Video presentations for the home consumer market can be composed of multiple clips (sequences of frames) which have been copied and stored as a single compressed video stream on an optical disk [Robi90], [Luth89]. This copy and paste approach requires duplication of a video clip each time it occurs in the sequence.

In many applications clips occur only once so that just one copy of the original source is made. For other applications such as a motion analysis study or an artistic work, clips may be repeated many times (possibly with overlaid modifications) so that the sequentially stored data stream may be many times larger than the original clips. Additionally, the requirement that data be sequenced and stored before it is presented prohibits applications with loops in the data path since these may define a sequence of unbounded length.

Another limitation of presequencing is that for presentations that are defined interactively, to be viewed only once, the intermediate sequencing and storage phase would appear to the user as an unnecessary delay.

3.2.2. Dedicating resources to avoid contention.

Dedicating storage resources for a particular set of accesses avoids delays due to multi-tasking and sharing conflicts. Variations on this technique range from assigning a single use for a disk drive at design time to short term allocation of a resource for exclusive use at run time. It effectively eliminates sharing, making delays simple and predictable.

All video display systems surveyed reserve the video storage device and buffer space during transfers in order to meet uninterrupted throughput requirements [Luth89], [Fox89], [Rang91]. Coarse resource reservations such as this greatly simplify the verification of a system's ability to meet application latency constraints.

Avoidance of contention is necessary in general, in order to guarantee performance. However, it is also possible to provide dynamic scheduling of resource allocations without violating previous commitments [Stan90], [Ande91].

3.2.3. Distributing data to increase throughput.

It is possible that a particular storage device cannot meet the throughput requirements of a CLSA application. *Striping* data across multiple storage servers (or disks) has been proposed in the Swift storage architecture as a technique for meeting high throughput requirements [Cabr91]. This technique can be exploited at the application level, or within the storage system according to application requirements.

Striping relies on presequencing data as described above, but in this case the data is also distributed for parallel access, effectively reducing the latency for large data blocks. This technique is useful primarily for periodic, sequential access patterns. Note that striping can not reduce latency for a given access below that of a single disk. In order to meet latency constraints that are smaller than the individual device access times, prefetching is still needed.

3.3. Prefetching

The following prefetching schemes all use accurate advance knowledge of the access stream to one degree or another. Prefetching is the only general means of accommodating large latencies in a CLSA application.

3.3.1. Massive prefetching.

Real-time computing has traditionally dealt with critical applications that cannot afford to miss deadlines. When storage requirements were modest, designers have chosen to eliminate the uncertain latencies of secondary storage by pinning everything in memory. Loading all code and data for an application before execution can be viewed as massive prefetching with minimal knowledge of the access pattern and total disregard for the storage latency (since it is incurred before the application starts). Even when an application does not explicitly access secondary storage, virtual memory paging mechanisms can introduce storage delays at unpredictable times. Real-Time Mach [Toku90] provides a kernel call, vm_wire, which allows a real-time thread to fix portions of its address space in memory at run time.

In a paper on real-time extensions to Mach, Nakajima, et. al. also use vm_wire to avoid unpredictable paging [Naka91]. Their motivation was the support of multimedia applications, in particular, a MIDI (musical instrument digital interface) handler for an interactive music system. However, the applications discussed did not have large storage requirements.

The Spring Kernel [Stan88] is designed to provide the basis of a flexible operating system for next generation, critical, hard real-time systems. Tasks that are deemed critical or essential are made memory resident in order to make execution times predictable. Their assumption is that violating the deadline of a critical task will have catastrophic consequences for the system (if not the world) and that virtual memory paging is not predictable. This project also offers other approaches for scheduling real-time storage tasks (discussed below under dynamically scheduled prefetching).

Of course the problem with preloading all data in memory is that memory is expensive and many multimedia applications have requirements that exceed any reasonable memory size. Furthermore, preloading massive amounts of data may introduce an unnecessary and unacceptable delay in itself. To use memory efficiently, knowledge must be exploited not only of *what* data is needed but *when* it will be needed.

3.3.2. Rehearsed prefetching.

An expedient method for synchronizing the access of stored data is to determine empirically how long the access takes and then to code this information into the application. If two objects are requested from separate storage units and one is available T seconds before the other, then the application can be modified to request this object T seconds later so that both objects will be available simultaneously when the application runs again. This approach assumes that the application and its underlying storage system are executing deterministically in real-time and that the application can be tuned by repeated rehearsals.

This approach is particularly tempting to the user of some multimedia authoring systems such as MacroMind Director [Macr90] and Athena Muse [Hodg89] where the user can attach objects to a timeline. These authoring tools execute on single user computing systems so that there is no interference from other users. The timeline itself is a valuable specification abstraction independent of its ability to tune a presentation to match storage latencies.

Of course, if other processes may interfere or storage access times are not repeatable, this approach has little or no value. Rehearsed prefetching can meet the requirements of scheduled, scripted access patterns only when storage resources have been dedicated in a manner that makes latencies simple and repeatable. While a presentation can be tuned to a particular storage system it will not be portable to others without retuning. A final major criticism is that many applications cannot tolerate a delay while the latencies are empirically determined.

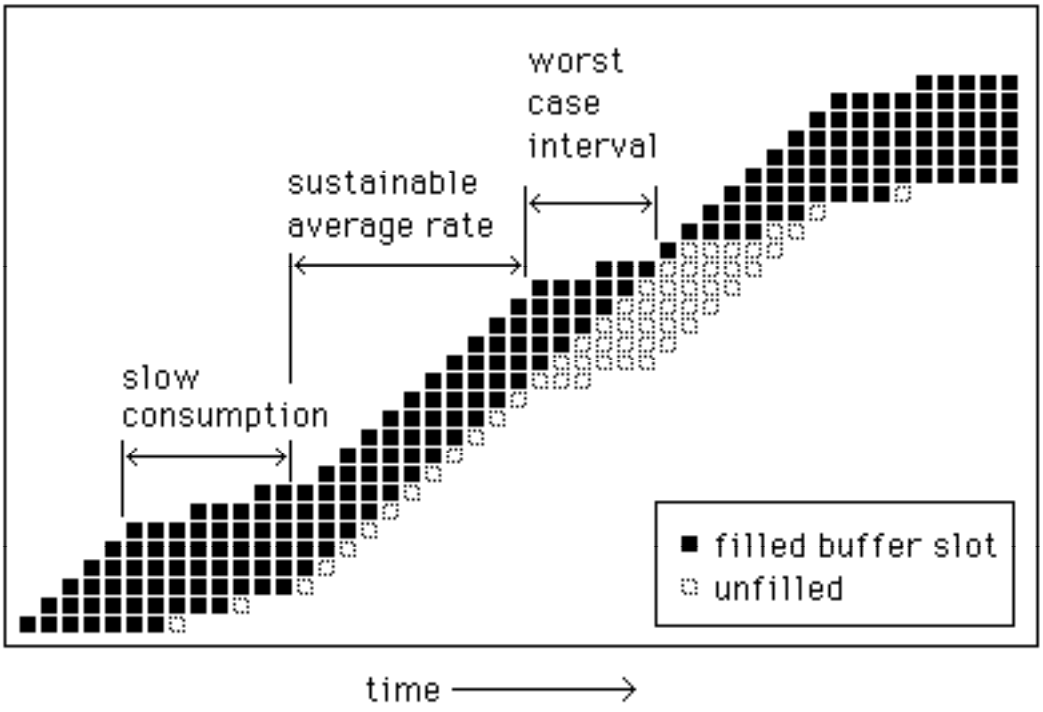


Figure 6. Greedy sequential prefetching with N = 6.

For the worst case interval, Δt = 6, D = 1, F = 1/3

3.3.3. Greedy prefetching.

When the sequence of data locations to be accessed is known the next data can be prefetched as soon as memory becomes available. This can be called the greedy approach to prefetching. The greedy prefetching approach requires knowledge of the access sequence, the average demand rate and the size of buffer needed to keep ahead of worst case

bursts in demand. If N is the size of the buffer in blocks and F and D are the rates at which blocks are prefetched and consumed respectively, then to assure that fetching stays ahead of demand, the following equation must hold:

$$\text{for every interval } (t_1,t_2): \quad D(t_2 - t_1) < F(t_2 - t_1) + N$$

This condition is illustrated in Figure 6. Note that if either of the rates F or D vary significantly, this condition may require a very large buffer size N.

Examples of this approach abound since the access pattern is readily identified in many applications. A few multimedia presentation architectures are discussed below to illustrate the technique.

The mechanism for presenting compressed video data from an optical disk drive using an IBM PC/AT with a Digital Video Interactive (DVI) video card involves some simple real-time scheduling extensions to MS-DOS and real-time threads that prefetch data from the disk to a fixed size buffer in video RAM. This design is predicated on the disk and cpu being dedicated for the video display task, guaranteeing sufficient throughput.

Another architecture specifically designed for studying storage systems that support video presentations has been developed at UCSD [Rang91]. Strand and rope abstractions are defined for synchronizing multiple streams of video and audio data. Rangan, et. al., note that their disk access library prevents them from playing a sequence of strands without a pause in between to close one strand and open the next. They plan to experiment with resource allocation to address this problem.

MediaView [Phil91] is a multimedia document editing application on the NeXT computer that integrates sound, animations and full motion video along with other media. This system supports video display from a dedicated device (e.g., a laserdisc player) using the greedy prefetching approach. Although the developers plan to support basic editing operations on video from a computer disk, there is currently no provision for synchronizing multiple real-time streams.

Greedy prefetching is most useful for periodic, sequential access patterns, where latency prediction is simple. When the worst case variation in the access or fetch rates is great, the buffer space required may be prohibitively large. If demand is not periodic, then a buffer sized to meet peak demand will be wasting scarce memory during the lulls. Multiple streams which are synchronized so that their peak demands do not coincide must still compete for buffer space based on their peak needs. A good analogy to illustrate the inefficiency problem of the greedy approach is to compare traditional manufacturing inventory approaches with Just In Time (JIT) scheduling [Voss88]. When manufacturers attempt to maintain large enough inventories of parts to handle bursts in consumption rates, they find that storage expenses and retrieval costs are high. JIT exploits more accurate knowledge of the future consumption rates in order to

move just enough of the parts to the consumer just when they are needed so that buffering resource requirements are minimal.

3.3.4. Dynamically scheduled prefetching.

The most efficient use of system resources for CLSA applications can occur if complete knowledge is available of when and what accesses are required by the application and how long storage access operations take. As discussed in Section 5, real-time scheduling techniques can find a near-optimal schedule (if one exists) that avoids resource conflicts and satisfies deadlines. This approach is attractive because it can enable applications on some platforms that would not otherwise be possible. Compared with the greedy prefetching approach, the use of more information about timing allows a smaller buffer space to be used. One way that buffer requirements can be reduced is that a scheduling function can see when worst case latency and access rates do not coincide. Another, is that during periods of low use, the space can be shared with other scheduled users.

Next generation real-time systems are expected to include tasks with specific resource needs that must be scheduled dynamically with timing guarantees [Stan88]. Although resource scheduling is an active research area for real-time systems, we are not aware of any systems that guarantee better than worst case timing constraints on general storage access.

A limited capability for scheduling prefetches is provided in a prototype client-server architecture for audio and telephony at Digital Equipment Corporation [Ange91]. An example is given of an answering machine, consisting of distributed logical devices that are controlled by a single command queue. When a play command is issued, a player device will not be able to retrieve and present the audio data instantaneously. Consequently, to support seamless transitions between sequential play commands, the protocol allows the player to inform the command queue of the time that the last sample will be played. The command queue can then pre-issue the next play command (if any), specifying that same time for the next play to start. This mechanism effectively communicates the application's synchronization constraints to the player which can then prefetch the data. Apparently, the player device is only given one command in advance at most, and the mechanism for prefetching is not specified.

Dynamically scheduled prefetching can use accurate information from scheduled, scripted applications and predictable storage systems to plan for the efficient use of resources that meets all constraints. The major limitations of the technique are that it is sensitive to both the number and the uncertainty of the constraints. The problem of scheduling a set of tasks with time and resource constraints is known to be NP-complete [Lens77]. While effective heuristic algorithms exist for this problem [Zhao87], they are sensitive to the uncertainty in task completion times. Worst case latency estimates can be so large as to make schedulability analysis impracticable. Complex hierarchical storage architectures [Hoga90] [Isra91] [Mill90], sharing and distribution all increase the variations in storage access latency and all are increasingly common.

The scheduling algorithms may themselves consume significant cpu resources. Some critical real-time systems partition the scheduling functionality to be performed on another processor so as not to interfere with the execution of previously guaranteed tasks [Stan90].

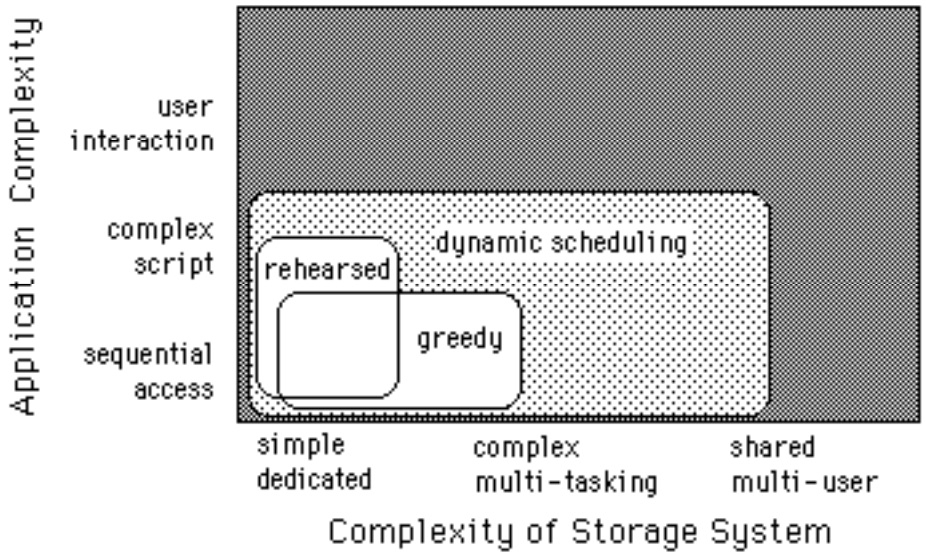


Figure 7. Generality of prefetching approaches.

4. Requirements for a More General Solution

With the exception of dynamically scheduled prefetching, the existing approaches for meeting CLSA requirements all make restrictive assumptions about access pattern and storage resources. A more general solution will support complex access to storage which is shared by multiple users (see Figure 7). One such solution might be to provide a storage manager which handles all access to a set of storage resources. The storage resources include at least one level of fast storage that caches data for which a latency constrained demand is believed to be imminent. As in the Spring Kernel [Stan90], a planning scheduler takes requests for future storage accesses and attempts to schedule them with respect to the current set of guaranteed tasks. A CLSA application may present a complex script to the storage manager, which in turn either guarantees or rejects it. A guarantee implies that a subsequent scheduled demand event will be satisfied by data which is ready in the cache.

Three hard problems in implementing such a solution can be identified:

- Finding and guaranteeing a feasible schedule for storage access tasks.
- Ensuring the availability of shared resources.
- Making application script information available to the scheduler.

The difficulties can be reduced by identifying script transformations that simplify while making only pessimistic adjustments to timing constraints. In addition, it should be possible to recognize familiar script patterns and use existing techniques to guarantee timely access. For example, greedy prefetching with an appropriately sized buffer can be used for periodic sequential access. Definition of other common access pattern types may allow a storage manager to schedule complex scripts using coarse abstractions of data access tasks.

The next three sections survey work that is related to solving these three problems.

5. Applicability of Real-Time Scheduling Techniques

Real-time scheduling research has focused primarily on scheduling the cpu in order to provide some level of guarantee for meeting task deadlines. The assumption was that storage access (typically only main memory) was an undifferentiated part of a task's computation time. For CLSA systems, access times are highly variable and storage access tasks need to be scheduled at a finer granularity in order to use our resources efficiently.

The Spring Kernel provides dynamic scheduling of new real-time tasks in parallel with the execution of previously guaranteed tasks [Stan90]. While designed for a multiprocessor and multi-node architecture, the principle feature of its scheduling approach is a functional partitioning of cpu and other resources between a *planning scheduler* and the *dispatching and execution* of guaranteed tasks. At any time, the system scheduler has knowledge of the currently executing set of guaranteed tasks, their resource requirements and worst case execution times. When a new task arrives, the scheduler uses a heuristic algorithm to attempt to schedule it while avoiding resource conflicts. If a feasible schedule is found, the new task is added to the guaranteed set and the old schedule replaced with the new. The designers of the Spring Kernel acknowledge that it was necessary to solve a number of race conditions to make the algorithm work.

The approach used to provide an on-line guarantee for tasks has a number of distinctive characteristics:

- By avoiding resource contention during task execution, predictability is enhanced and context switching is minimized.
- The system can offer early notification when a task cannot be guaranteed, allowing error handling or alternative scheduling before the task's deadline arrives.
- If a task finishes early, it is possible to reclaim the unused resources.

Two main concerns arise regarding the suitability of the Spring Kernel approach for CLSA applications. First, the task description does not include an earliest start time, implying that tasks will be dispatched as quickly as possible without regard to an application's schedule. The second concern regards an assumption that the number of guaranteed tasks is small. In complex scripted multimedia presentations, the number of storage access tasks to be scheduled can be very large and may swamp the capabilities of the algorithm used in the Spring Kernel. The complexity of this heuristic algorithm for scheduling a set of *n* tasks in a system having *r* resources is $O(rn^2)$ [Zhao87].

Blake and Schwan [Blak91] report on another dynamic scheduler for less-critical and low frequency tasks.[1] Their scheduler uses a bin-packing approach and is designed for a multiprocessor system to provide on-line guarantees (if possible) for dynamically occurring tasks. Requests for scheduling include a process' deadline, start time, execution time and resource constraints. As in the Spring Kernel, resources may be allocated exclusively by the scheduling process to avoid contention. Scheduling of periodic processes as a group and processes with precedence constraints are also supported. The bin-packing approach seems more appropriate for the advance reservation needs of a scripted CLSA application but is even more likely to be overwhelmed by large scheduling problems. Blake and Schwan claim only that the scheduling overhead is reasonable for moderate system loads of ten different deadline bins per processor.

Numerous algorithms have been described to perform schedulability analysis for real-time system design. Scheduling with resource requirements has been considered as early as 1980 by Leinbaugh [Lein80], but only for static analysis at design time. An interactive schedulability analysis tool has been described [Toku88] that can verify if a set of tasks will meet their deadlines under the rate monotonic scheduling algorithm [Liu73]. Designed for existing real-time systems, it does not consider the resource requirements of storage access tasks.

The current state of research in integrated cpu and resource scheduling indicates that designing a dynamic scheduling approach to complex CLSA applications is a difficult problem. CLSA applications may include hundreds of coarse synchronization constraints, each of which require computation of storage access, buffering and cpu resource requirements. The real-time scheduling techniques described above need to be evaluated with such large simultaneous scheduling problems in order to judge their applicability. If they are to be of use, they must be incorporated into real-time operating systems in the general purpose computing environments where multimedia CLSA applications occur. More work is also needed to provide automatic means for deriving timing constraints for storage access tasks from the high level constraints of a CLSA application.

---

[1] Very critical tasks are statically scheduled, as in the Spring Kernel.

6. The Value of Resource Reservation Protocols

One key to the scheduling algorithms just discussed was the ability to avoid contention for resources. It is important that the memory and other resource needs of a task be reserved in advance if the task is to execute predictably according to its schedule. With shared resources, such as a remote storage server, requests for reservations from different users may conflict. Some protocol is required to negotiate a guarantee for service.

The Session Reservation Protocol (SRP) [Ande90] allows peer computer nodes to reserve resources such as CPU and network bandwidth in order to transmit data streams with real-time performance guarantees. A data stream may traverse several hosts between a source and a sink, requiring the establishment of an end-to-end session including all the resources involved on each host. SRP operates by first establishing a transport-level connection between the source and sink clients. The performance goals for the data stream must be agreed upon and a session ID obtained. The source client reserves the local resources that it needs and then calls the local SRP module to reserve the network resources. The SRP modules on this and subsequent nodes in the communication pathway reserve the communication resources until the session request is forwarded to the sink client. At the sink, any remaining needed resources are reserved. As the session request moves from the source to the sink, pessimistic resource estimates are used to guarantee that the performance goals can be met. If the sink client determines that guarantees exceed the requirements, then the excess is passed back toward the source, and reservations are relaxed accordingly at each node according to specified cost functions.

SRP assumes that clients know their workload (the bandwidth and burstiness of a data stream) in advance. A client requests a session with a resource by specifying the workload. The resource, in turn, provides a bound on the delay it will impose. The client can decide if this delay is acceptable.

Recent work by Anderson, et. al. extends the use of session and workload abstractions for real-time scheduling guarantees in the Continuous Media File System (CMFS) [Ande91]. CMFS can accept or reject a request for a continuous media session based on the workload of the request, knowledge of existing sessions and bounding functions for storage access times. Simulation results are given [Ande91], to compare disk scheduling policies and buffer requirements. The CMFS uses greedy prefetching into a FIFO buffer to satisfy the latency constraints within continuous media streams. Synchronization between multiple streams is possible only by establishing all sessions in advance and blocking each until the synchronization time.

While the Session Reservation Protocol provides essential throughput guarantees, it does not support the anticipation of session startup latencies. Dynamic scheduling of remote storage access tasks will need a protocol which provides advance reservations and delay guarantees in order to satisfy hard latency constraints. It is not clear that existing scheduling algorithms can be extended effectively to incorporate such a protocol.

7. Scripting Languages

Many multimedia presentations are completely deterministic in their access to stored data. In order for a scheduling algorithm to take advantage of this feature, the application must provide some sort of a script. This section surveys, briefly, a few of the scripting languages which have been proposed for multimedia editing.

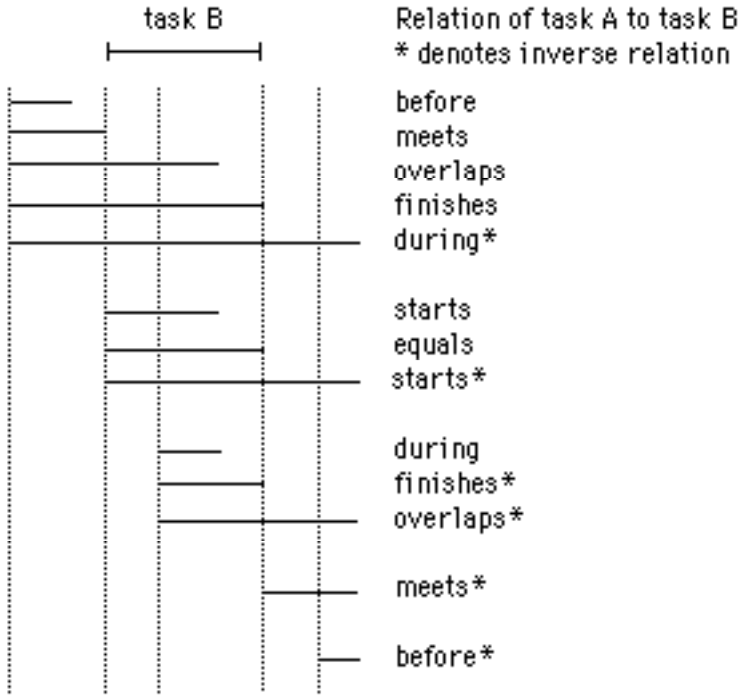


Figure 8. Temporal relations in Object Composition Petri Nets.

Little and Ghafoor have noted that the scheduler for the access and presentation of objects must have access to a schema specifying the temporal relationships of the objects to be presented, the media types of the objects, the required quality of service (QOS) for continuous media objects, and the storage location of the selected objects [Litt90a]. They propose a technique for specifying coarse synchronization relationships between multimedia presentation tasks using Object Composition Petri Nets (OCPN) [Litt90b]. The OCPN allows an arbitrarily complex composition of the seven basic relations listed in Figure 8. (Exactly 13 relations are possible between two intervals, but six are inverses.) Complex OCPNs are constructed pairwise by specifying two objects, the duration of each, the temporal relation between them and the value of a delay parameter. Every OCPN can be viewed as another object with known duration for recursive composition.

Objects with the continuous media type include audio and video streams that are composed of discrete samples. For these streams, the presentation schema includes a tuple of quality of service (QOS) parameters which include speed ratio, utilization, skew, jitter, and reliability. The reliability parameter consists of Bit Error Rate (BER) and Packet Error Rate (PER) tolerances.

The Athena Muse project [Hodg89], provides four distinct representational approaches for specifying an interactive multimedia learning environment: directed graphs, multidimensional spatial frameworks, declarative constraints, and a procedural language. The directed graphs are useful for hypermedia style navigation. In contrast, the spatial frameworks allow both specification of window positions and placement of objects on a presentation timeline. A timeline can be used to synchronize many objects including still images, video and audio tracks. Declarative constraints, in the Muse system, are limited to bidirectional equality relations. The example they give is of a scroll bar that is constrained to represent the current position in the display of a timeline. When the scroll bar is moved, the view from the timeline is updated and vice versa. Finally, the procedural language allows arbitrary computations to be embedded in the production.

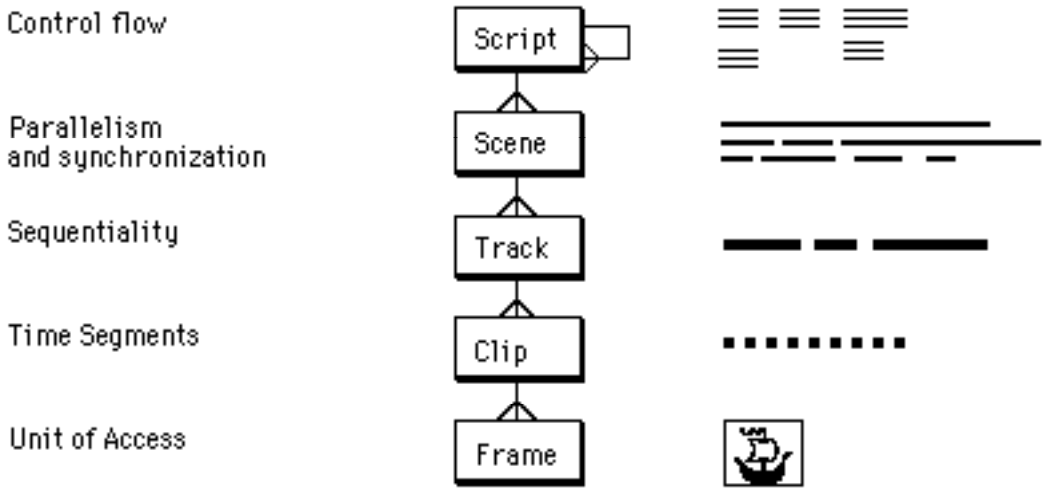


Figure 9. Pro750 GUI script elements, from [Skar90].

The Pro750 Graphical User Interface describes a high level scripting language [Skar90] that is similar to that of the Athena Muse project. Figure 9 illustrates the basic elements of a script. The simplest element is a frame which, in the case of video, is a single still image. The next level of script elements is the clip, which has a time interval associated with it. A video clip defines a collection of frames that are to be displayed during the time interval. Clips

of the same media type may be composed in series to form a track. A track can be assigned to a single output resource such as a display window or a speaker. Multiple tracks may be combined in parallel to form a scene. Finally, a script is composed of scenes and other sub-scripts. A script has an implicit timeline to which all its components are attached. The timeline begins at zero, so that if the script were to be scheduled at some real system time T then all events within the script would be offset logically by the value of T.

8. Conclusion

This paper has identified a class of Constrained Latency Storage Access (CLSA) applications that require both large amounts of storage and guarantees for short latencies. A broad range of approaches to meeting the requirements of CLSA applications were surveyed in Section 3. The limitations of current approaches indicate that the technology does not yet exist to support complex CLSA applications on general purpose storage architectures. A variety of good approaches exist for meeting throughput and latency requirements for sequentially accessed data, including dedication of resources, presequencing, and greedy prefetching. However, none of the current approaches support the automatic anticipation of scripted data access needs such as are required for interactive editing and playback of video segments.

There seem to be two main reasons why the unique character of the CLSA problem has not been identified previously in the literature. First, most real-time systems in the past have had modest storage requirements, allowing designers to fix all code and data in main memory. Second, in order to prefetch data it is necessary to have knowledge of what data will be needed and when. Most computer applications do not have a known storage access pattern and existing latency minimization techniques that guess this information do not guarantee the near 100% accuracy required for hard latency constraints [Palm91], [Krat90], [Bela66].

Both of these norms are changing. As the power of processors and networks has grown, so has the volume of the data being manipulated. General purpose computers are also increasingly being used for the mundane task of presenting information where the timing and location of data accesses can be known well in advance. For the first time, real-time applications that have both a large appetite for storage and a great deal of predictability are becoming common.

Dynamic scheduling and reservation of resources represents a relatively untried, knowledge intensive approach to CLSA applications. A planning scheduler can prefetch in order to meet synchronization constraints while avoiding conflicts with other guaranteed tasks. The access location and timing information required by the scheduler may be derived from high level application scripts. Resources can be used more efficiently when they are reserved only as required to satisfy

specific application constraints. Despite its promise, the dynamic scheduling approach requires further progress in several research areas:

- Efficient real-time scheduling of large task sets with resource constraints.
- Protocols for advance reservation of resources.
- Derivation of access timing and location requirements from high-level scripts.

While existing systems do not support the general class of CLSA applications they do support useful subsets, e.g. presentation of contiguous real-time data streams. Future work can extend the capabilities of these systems by incorporating resource reservations and dynamic scheduling algorithms that make increasing use of application scripts.

The major contribution of this paper is in characterizing the CLSA problem and documenting the partial solutions that exist in the current literature. It is hoped that this work will provide good direction to researchers who would like to contribute to more general solutions.

9. Acknowledgements

This paper owes much to discussions with Dave Maier at OGI. We would also like to thank Dave Maier, Jim Hook and David Novick for their careful reading and critique of the paper.

References

Abbo88. Abbott, R. and Garcia-Molina, H. Scheduling Real-time Transactions. *SIGMOD Record 17*, 1 (March 1988), 71-81.

Ande90. Anderson, D.P., Herrtwich, R.G., and Schaefer, C. SRP: A Resource Reservation Protocol for Guaranteed- Performance Communication in the Internet. Tech. Rept. TR-90-006, ICSI, February, 1990.

Ande91. Anderson, D.P., Osawa, Y., and Govindan, R. Real-Time Disk Storage and Retrieval of Digital Audio and Video. Tech. Rept. UCBTR-91-646, UCB, 1991.

Ange91. Angebranndt, S., Hyde, R.L., Luong, D.H., Siravara, N., and Schmandt, C. Integrating Audio and Telephony in a Distributed Workstation Environment. In *Proceedings of the Summer 1991 USENIX Conference*, 1991, pp. 419-435.

Bela66. Belady, L.A. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal 5*, 2 (1966), 78-101.

Berr90. Berra, P.B., Chen, C.Y.R., Ghafoor, A., Lin, C.C., Little, T.D.C., and Shin, D. Architecture for Distributed Multimedia Database Systems. *Computer Communications [UK] 13*, 4 (May 1990), 217-231.

Blak91. Blake, B.A. and Schwan, K. Experimental Evaluation of a Real-Time Scheduler for a Multiprocessor System. *IEEE Transactions on Software Engineering 17*, 1 (January 1991), 34-44.

Cabr91. Cabrera, L.F. and Long, D.D.E. Exploiting Multiple I/O Streams to Provide High Data Rates. In *Proceedings of the Summer 1991 USENIX Conference*, 1991.

Chri86. Christodoulakis, S. and Faloutsos, C. Design and Performance Considerations for an Optical-Disk Based, Multimedia Object Server. *IEEE Computer 19*, 12 (Dec 1986).

Cohe89. Cohen, E.I., King, G.M., and Brady, J.T. Storage hierarchies. *IBM Systems Journal 28*, 1 (1989).

Fox89. Fox, E.A., Hix, D., Schwartz, E.E., Siochi, A., Koushik, P., and Inman, D. Interactive Digital Video Authoring and Prototyping. Tech. Rept. TR 90-13Virginia Polytechnic Institute and State University, December, 1989.

Fren89. Frenkel, K.A. The Next Generation of Interactive Technologies. *Communications of the ACM 32*, 7 (July 1989), 872-881.

Hodg89. Hodges, M.E., Sasnett, R.M., and Ackerman, M.S. A construction set for multimedia applications. *IEEE Software* (January 1989), 37-43.

Hoga90. Hogan, C., Cassell, L., Foglesong, J., Kordas, J., Nemanic, M., and Richmond, G. The Livermore Distributed Storage System: Requirements and Overview. In *Digest of Papers, Tenth IEEE Symposium on Mass Storage Systems, May 7-10, 1990*, IEEE Computer Society, 1990, pp. 6-17.

Isra91. Israel, R.K., Foster, A.W., Taylor, A., Taylor, T.M., and Webber, N. Evolutionary Path to Network Storage Management. In *USENIX Winter '91, Dallas, TX*, USENIX, 1991, pp. 185-197.

Krat90. Kratzer, K., Wedekind, H., and Zörntlein, G. Prefetching - a performance analysis. *Information Systems 15*, 4 (1990), 445-452.

Lein80. Leinbaugh, D.W. Guaranteed Response Times in a Hard-Real-Time Environment. *IEEE Trans. Software Eng. SE-6*(January 1980).

Lens77. Lenstra, J.K., Rinnooy, A.H.G., and Brucker, P. Complexity of Machine Scheduling Problems. *Annals of Discrete Mathematics 1*(1977).

Litt90a. Little, T.D.C. and Ghafoor, A. Network Considerations for Distributed Multimedia Object Composition and Communication. *IEEE Network Magazine* (November 1990), 32-49.

Litt90b. Little, T.D.C. and Ghafoor, A. Synchronization and Storage Models for Multimedia Objects. *IEEE J. Sel. Areas Commun. (USA) 8*, 3 (April 1990), 413-427.

Liu73. Liu, C.L. and Layland, J.W. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *JACM 20*, 1 (1973), 46-61.

Luth89. Luther, A.C. *DVI: Digital Video in the PC Environment*, McGraw-Hill, New York (1989).

Macr90. *MacroMind Director 2.0.1*, MacroMind,Inc., 410 Townsend, Suite 408, San Francisco CA 94107, 1990.

Mill90. Miller, S.W. Review of IEEE-CS MSS Reference Model. In *Tenth IEEE Symp. on Mass Storage Systems*, 1990.

Naka91. Nakajima, J., Yazaki, M., and Matsumoto, H. Multimedia/Realtime Extensions for Mach Operating System. In *Proceedings of the Summer 1991 USENIX Conference,* 1991.

Palm91. Palmer, M. and Zdonik, S. *Fido: a cache that learns to fetch*. 1991, To appear in 1991 VLDB conference proceedings..

Phil91. Phillips, R.L. MediaView: A General Multimedia Digital Publication System. *Communications of the ACM 34*, 7 (July 1991), 75-83.

Rang91. Rangan, P.V., Burkhard, W.A., Bowdidge, R.W., Vin, H.M., Lindwall, J.W., Chan, K., Aaberg, I.A., and Yamamoto, L.M. A Testbed for Managing Digital Video and Audio Storage. In *Proceedings of the Summer 1991 USENIX Conference,* 1991.

Robi90. Robinson, P. The Four Multimedia Gospels. *BYTE (USA) 15*, 2 (Feb 1990).

Sing88. Singhal, M. Issues and Approaches to Design of Real-Time Database Systems. *SIGMOD Record 17*, 1 (March 1988), 19-33.

Skar90. Skarbo, R.A. A High Level Description of the Pro750 GUI. Tech. Rept. SWTD PNB/152 (Section 5.0)Intel SW Technology Development, September, 1990.

Stan88. Stankovic, J.A. and Zhao, W. On Real-Time Transactions. *SIGMOD Record 17*, 1 (March 1988), 4-18.

Stan90. Stankovic, J.A. and Ramamritham, K. *The Spring Kernel: A New Paradigm for Next Generation Hard Real-Time Systems*. September 1990, Unpublished copy obtained from the author.

Toku88. Tokuda, H. and Kotera, M. Scheduler 1-2-3: An interactive schedulability analyzer for real-time systems. In *Proceedings of Compsac88,* October 1988.

Toku90. Tokuda, H., Nakajima, T., and Rao, P. Real-Time Mach: Towards a Predictable Real-Time System. In *Mach Workshop,* USENIXAssociation, 1990.

Voss88. *Just-in-Time Manufacturing*, Springer-Verlag (1988).

Wede86. Wedekind, H. and Zoerntlein, G. Prefetching in realtime database applications. *ACM SIGMOD Record. 15*, 2 (June 1986), 215-226.

Zhao87. Zhao, W. and Ramamritham, K. Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints. *Journal of Systems and Software* (August 1987).