An Event-Based Model for Continuous Media Data on Heterogeneous Disk Servers*

K. Selçuk Candan, **Eenjun Hwang**, and **V.S. Subrahmanian**

Abstract

Over the last few years, there has been intense work on the problem of retrieval of continuous media (CM) data from disk. However, no single unified framework exists within which such retrieval problems can be studied. In this paper, we first propose a *formal model* for CM-data retrieval from heterogeneous disk servers. This model can be used to characterize CM-data retrieval problems *independently* of how data is laid-out on disk, and what objectives (e.g. minimize client delay, maximize buffer utilization, etc.) the system manager is interested in. We then show how using this formal model, we can neatly define what it means to optimally handle events that occur in movie-on-demand (MOD) systems. Examples of such events include new clients entering the system, old clients leaving the system, continuing clients doing pause, rewind and fast-forward operations. Multiple events may occur simultaneously and we show how such events trigger state-transitions in the system. We then develop an algorithm called the **QuickSOL** algorithm that handles events occurring in MOD systems. This algorithm works in two phases: in the first phase it quickly finds a way of handling as many of the events occurring at time t as possible. In the second phase, it optimizes the solution found in the first phase. The advantage is that the algorithm can be interrupted anytime after the first phase is completed. We report on experiments showing that **QuickSOL** works well in practice.

1 Introduction

During the last few years, the problem of efficiently retrieving continuous media data (CM-data, for short) from disk has been recognized as a major problem that must be solved if applications such as video-on-demand are to become a commercial reality. Towards this end, several authors [2, 17, 18, 21, 24] have proposed frameworks for organizing CM-data on disk – furthermore, a wide variety of impressive experimental results [3, 10, 8, 12, 15] have been reported in the literature. In this paper, we will propose a *unified framework* for the modeling and evaluation of CM-disk servers. This unified model will have the following features, many of which are new:

1. We model movie-on-demand (MOD) systems where the data is located across **multiple, heterogeneous disk servers.** The assumption that movie-on-demand systems will use homogeneous

*This work was supported by the Army Research Office under Grant Nr. DAAH-04-95-10174, by the Air Force Office of Scientific Research under Grant Nr. F49620-93-1-0065, and by ARPA/Rome Labs contract F30602-93-C-0241 (ARPA Order Nr. A716). Authors' Address: Department of Computer Science, University of Maryland, College Park, MD 20742. {candan,hwang,vs}@cs.umd.edu

disks is unrealistic for several reasons: first, at a strategic level, most companies balk at having a single supplier for a given resource (e.g. disk) – hence, they are likely to obtain supplies from a variety of disk vendors, all of whom provide disks with varying characteristics. Second, at various points in time, they are likely to "upgrade" their disks to take advantage of advances in disk technology. Third, they may wish to re-use some of their old disk servers for financial reasons. Our analytic model can be used to model any set of heterogeneous disks/servers as long as some information about the characteristics of these servers can be provided to us.

2. Most existing disk scheduling algorithms are tightly-coupled to a *specific* goal – e.g. maximize disk utilization[15, 12], minimize jitter, minimize delay time maximize buffer utilization[15, 19, 12], to name a few. In this paper, we discuss which goals should be used to improve the efficiency of a video disk system, and we show how MOD systems can handle a combination of such goals. Our framework is robust in the sense that our algorithms are not strictly bound to the set of objectives chosen, but the system benefits from them. In other words, the MOD system manager may plug in different objective functions as time proceeds and evaluate the performance of his system.

3. Most MOD systems require real-time responses – hence, our algorithms are set up to work in two phases – a `FindSOLution` phase which is executed very fast, and an `OptimizeSOLution` phase that iteratively proceeds towards a better solution. Consequently, if a hard real-time deadline "comes up" during the `OptimizeSOLution` phase, our algorithm can be interrupted and will return the best solution found thus far. However, like any other system, it is always possible that a given set of clients may place such high demands on the MOD system that it is impossible to find solutions within the real-time deadline, in which case some clients may encounter service delays. This would occur if the real-time deadline comes up before the `FindSOL` phase is completed. *To our knowledge, this is the first provably correct, optimal algorithm for retrieving continuous media data from disk.*

4. Most existing algorithms assume that the CM-data being accessed is laid out on disk in a *specific way* – e.g. by striping[21, 18]. We expect (and hope !) that techniques to lay out data on disk will improve as time goes by. *To our knowledge, our algorithm is the first one that is designed to work with any placement mapping whatsoever and to consider such maps as part of the input* – a placement mapping specifies how data is laid out on disk.

5. We provide a formal definition of the **state** of an MOD system, and show how various operations (e.g. new client enters, old client exits, continuing client watches next block of movie, continuing client pauses the movie, continuing client rewinds or fast-forwards the movie) cause **state-transitions**. We formally define what states satisfy a set of events, and how to compute an "optimal" next state that satisfies the system constraints. It is entirely possible that there are many possible ways of handling the transaction(s) that occur at a given point in time. Each of these different ways may cause the MOD system to transition to a different new state. We develop evaluation functions that assess the "goodness" of a state to which the system may transition, and based on such evaluation functions, develop the notion of an *optimal* state to which the system may transition. *To our knowledge, we are the first to provide a formal, declarative, mathematical definition of what it means to optimally handle events in MOD systems.* The declarative, non-algorithmic, nature of this definition provides a formal specification that serves as a yardstick against which correctness, soundness, and optimality of algorithms may be measured.

Symbol	Meaning
`bnum`($\mathcal{M}_i$)	Number of blocks in movie $\mathcal{M}_i$
`buf`(i)	The total buffer space associated with disk server i
`cyctime`(i,t)	the total cycle time for server i at time t
`dtr`(i)	The total disk bandwidth associated with disk server i
`switchtime`(i,t)	the time required for disk server i to switch from one client's job to another client's job at time t
`cons`(i,t)	The consumption rate of client C_i at time t
`data`(i,t)	The event specification for the client C_i at time t
`timealloc`(i,j,t)	The time-slice allocated to client j at time t
`active`(t)	The set of all clients that are active at time t
`d_active`(i,t)	The set of all clients that have been assigned a non-zero time-allocation by disk server i
$\wp(\mathcal{M}_i, b)$	The set of servers that contain block b of movie $\mathcal{M}_i$ according to placement mapping $\wp$
$\mu_t(i)$	The set of servers handling requests by client C_i at time t
`bufreq`(j,i,t)	The buffer space needed at server i to match the consumption rate of client j
$\mathcal{S}(t)$	The state of a movie-on-demand system

Figure 1: Notation and terminology

6. We develop the **QuickSOL** algorithm that is guaranteed to be provably sound, complete, and optimal. This algorithm forms the heart of our system. **QuickSOL** may be used in conjunction with arbitrary placement mappings and arbitrary evaluation functions – it will still be sound, complete and optimal. *To our knowledge, our algorithm is the first one that can work with arbitrary state-evaluation functions plugged in to assess how best to handle new requests/transactions in MOD systems – furthermore, as previously algorithms for processing user requests had no attached notion of "optimal handling" of such requests, our algorithm is the first to do so. .*

7. Finally, we report on results obtained by a simulation of our algorithms using simulation parameters in [12, 15, 19, 20]. The experimental data allows us to derive specific numbers of clients that can be supported using **QuickSOL** given certain assumptions on disk server and client characteristics. Furthermore, the experimental algorithms allow us to assess the effectiveness of different evaluation functions.

2 Formal Analytic Model

Figure 1 provides an overview of the notation we use in this paper. This section provides a detailed explanation of the notation and terminology shown in the figure.

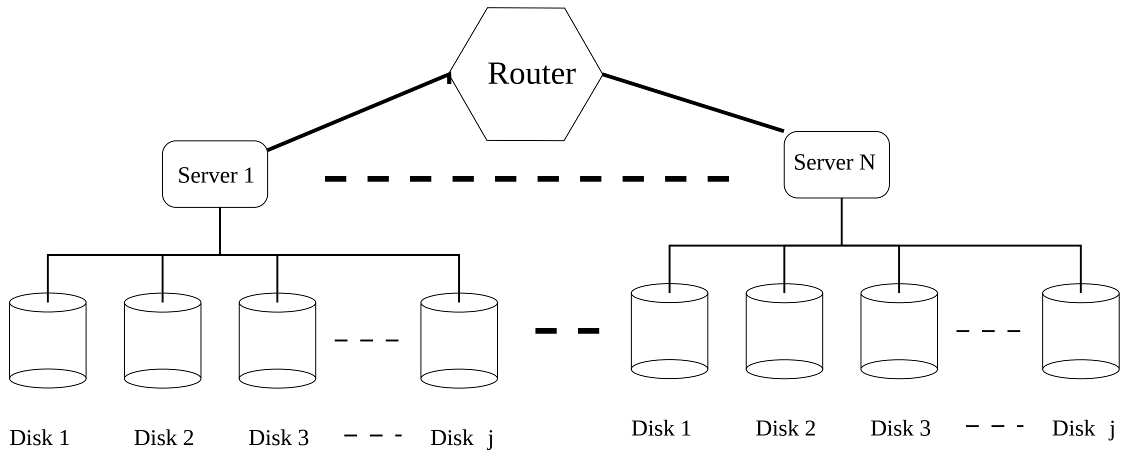

Figure 2: The server architecture

2.1 Characterization of Movies

Throughout this paper, we assume that there is a set $\mathcal{MOVIE} = \{\mathcal{M}_1, \ldots, \mathcal{M}_r\}$ of movies that we wish to store on disk. Each movie $\mathcal{M}_i$ has $\mathsf{bnum}(\mathcal{M}_i)$ "blocks." A block denotes the level of granularity at which we wish to store and reason about the media-data. For example, a block may be a single-frame (finest granularity) or a consecutive sequence of (100 frames).

2.2 What Needs to Be Modeled ?

When developing a formal model that characterizes the performance of a heterogeneous disk system for multimedia applications, we need to characterize the following aspects of the system:

1. The intrinsic characteristics of each disk server;
2. The intrinsic characteristics/capabilities of each client;
3. The relationship between the disk servers and clients;
4. The distribution of data across the disk servers.

- **Disk Server Characteristics:** Suppose we have N disk servers. Each disk server accesses a homogeneous disk array as shown in figure 2. Note that there is no loss of generality in making this assumption. If a disk server has n different types of disks in it, the disk server itself may be represented as n different servers, each serving disks of only one type. However, different disk servers may access different, heterogeneous disks. The characteristics of each disk server may be obtained from the characteristics of the disks in the disk array by using methods such as those of Ng. et. al. [12, 15]. These characteristics include:

  1. $\mathtt{buf}(i)$: the total buffer space associated with disk server i, for $1 \leq i \leq N$.

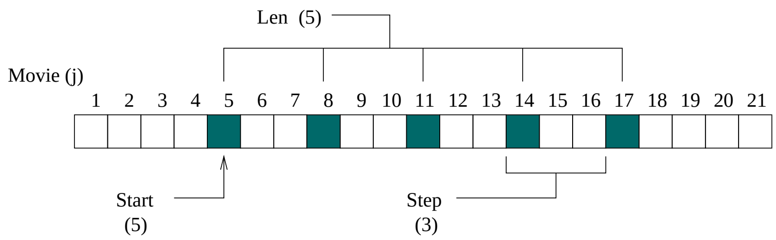

Figure 3: $\texttt{data}(i,t) = (\mathcal{M}_j, \texttt{5}, \texttt{5}, \texttt{3})$

2. $\texttt{cyctime}(i,t)$: the total cycle time[20, 19, 15] for a read-operation performed by disk server i at time t. Intuitively at any given point in time t, each disk server i has a "read-period" – slices of this read-period are allocated to the different clients (any time in the read-period not allocated to a client may be used for other operations such as pre-fetching[15, 10], etc.)
3. $\texttt{dtr}(i)$: the total disk bandwidth associated with disk server i.
4. $\texttt{switchtime}(i,t)$: this specifies the time required for disk server i to switch from one client's job to another client's job at time t.

- **Client Characteristics:** Different clients participating in an MOD system may have very different characteristics. The *client-pool* of a given MOD system is a set $\mathcal{CL} = \{C_1, C_2, \ldots, C_k\}$ – intuitively, suppose for instance that a major company such as Blockbuster Video has an MOD facility. Then $C_1, C_2, \ldots, C_k$ may be thought of as a list of all Blockbuster video subscribers, even though only a small fraction of these subscribers may in fact be viewing a movie at any given time.

  The characteristics of clients are captured by the following items.

  1. $\texttt{cons}(i,t)$: The consumption rate of client C_i at time t;
  2. $\texttt{data}(i,t)$: This is a quadruple $(\mathcal{M}_j, \texttt{start}, \texttt{len}, \texttt{step})$ saying that at time t, client C_i is accessing `len` blocks of movie $\mathcal{M}_j$, such that the individual blocks are separated from each other by $\texttt{step} - 1$ intermediary movie blocks.

  Intuitively, `step` is always 1 when normal viewing is considered; however when a user "pauses" the system, `step=0`. Similarly, if the user "fast-forwards" (resp. rewinds) then `step` equals the fast-forward factor (resp. rewind factor) which is typically greater than 1 (resp. less than 0). Thus, for instance, if $\texttt{data}(i,t) = (\mathcal{M}_j, \texttt{5}, \texttt{5}, \texttt{3})$, then this means that the user wants to see 5 "blocks" of the move $\mathcal{M}_j$, starting from block 5 at steps of 3 (i.e. the user is fast-forwarding with factor 3 in this case). Therefore, as shown in Figure 3, the user will see blocks 5,8,11,14 and 17, respectively.

  It is important to note that once the consumption rate of a client is fixed, this automatically places an upper bound on the number of blocks (`len` above) that the client can request. The exact constraint defining this depends on the block-size. In this paper, we assume that clients will always request a fewer number of blocks than the upper bound alluded to above.

- **Client - Server Characteristics:** In addition to characteristics that are intrinsic to either disks or clients, there are properties of a multimedia system that depend both upon disk characteristics as well as on client characteristics. Some of these are summed up below:

1. $\texttt{timealloc}(i, j, t)$: This specifies the time-slice (within the time period $\texttt{cyctime}(i, t)$ of disk server i) that has been allocated to client j at time t.

Definition 2.1 (Active Clients) A client C_j is said to be *active* at time t iff $(\exists i)\texttt{timealloc}(i, j, t) > 0$. $\diamond$

In other words, a client is active at time t just in case some disk server has allocated time to it. We use $\texttt{active}(t)$ to denote the set of all clients that are active at time t. We use $\texttt{d\_active}(i, t)$ to denote the set of all clients that have been assigned a non-zero time-allocation by disk server i.

The following axiom states that the cycle time of a server must be chosen so that within a cycle time all the clients assigned to the server can be serviced:

(CycTime Constraint) "At any given point in time, the cycle time of the server i is greater than or equal to the sum of the service times of the clients serviced by the server i."

$$(\forall 1 \leq i \leq N)(\forall j, t) \qquad \texttt{cyctime}(i, t) \geq \left(\left(\sum_j \texttt{timealloc}(i, j, t) + \texttt{switchtime}(i, t)\right)\right).$$

Next, we define a mapping which describes the way movie blocks are placed on the disk servers.

Definition 2.2 (Placement Mapping) A *placement mapping* ($\wp$) is a function that takes as input:

1. a movie-id, $\mathcal{M}_i$, and
2. a block number $1 \leq f \leq \mathsf{bnum}(\mathcal{M}_i)$

and returns as output, a subset of $\{1, \ldots, N\}$ where N is the number of disk servers. $\diamond$

Intuitively, if $\wp(\text{"The Rope"}, 300) = \{2, 4, 5\}$, then this means that block 300 of the movie "The Rope" is accessible via each of the disk servers $2, 4$, and 5. In particular, our model assumes that a given movie (or even certain blocks of a movie) can be replicated across multiple disk servers.

Note that the placement mapping can dynamically change over time due to disk administration purposes. In fact, our system reduces the needs for such dynamic changes by guaranteeing a balanced load distribution by splitting and switching clients, rather than the expensive operation of copying large amounts of media data from one disk to another. However, in some cases (like a disk crash), some dynamic placement modifications may be unavoidable. Our framework may be easily extended to handle such dynamic changes, by defining the placement mapping to be a *ternary* map (rather than a binary one), with the third item being a time-instant; thus, $\wp(\mathcal{M}_i, b, t)$ would specify the set of all servers at which block b of movie $\mathcal{M}_i$ is located at time t. As this paper already uses lots of symbols (cf. Figure 1), we will continue to use binary placement maps for the sake of notational simplicity.

Definition 2.3 (Client Mapping) A *client mapping*, μ_t, at time t is a mapping $\mu_t : \texttt{active}(t) \rightarrow 2^{\{1,\ldots,N\}}$. $\diamond$

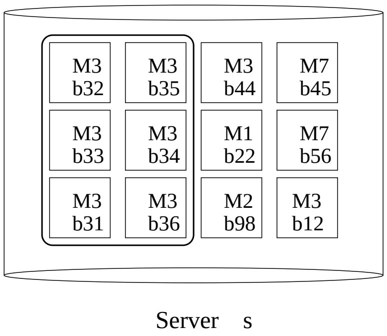

Figure 4: $segment(31, 36, s, M3)$

Intuitively, if $\mu_t(j) = S$, then at time t the client j is served by the servers S. For example, if $\mu_5(C_4) = \{1, 6\}$, then this means that at time 5 client C_4 is being served by both disk servers 1 and 6. In this case, the requested blocks are provided by servers 1 and 6.

The following axiom states that, if a client j is assigned to be processed by server i, then server i allocates a non-zero amount of time to the processing of client j's request:

(Allocation Constraint):

$$i \in \mu_t(C_j) \stackrel{def}{\Leftrightarrow} \texttt{timealloc}(i, j, t) \neq 0.$$

The following definitions describe the way movie blocks are clustered in the individual disk servers.

Definition 2.4 (Segment) Let b_i and b_j where $i \leq j$ be blocks of movie m. The blocks b_i and b_j define a segment for movie m in server s iff the following holds:

$$\forall(k)((i \leq k \leq j \rightarrow s \in \wp(m, k))\&(k = i - 1 \rightarrow s \notin \wp(m, k))\&(k = j + 1 \rightarrow s \notin \wp(m, k)))$$

We use $segment(i, j, s, m)$ to denote the segment of movie m defined by blocks b_i and b_j in server s. $\diamond$

Figure 4 shows $segment(31, 36, s, M3)$. Note, however, that there is no segment called $segment(33, 36, s, M3)$, because $segment(31, 36, s, M3)$ already includes the blocks from 33 to 36.

Thus far, we have defined various technical notions. We are now ready to define some constraints that "tie" together, the characteristics of the servers participating in an MOD system, and the clients using the system at a given time instant t.

Definition 2.5 (Placement Constraint) A *placement constraint* (PC) says that the client mapping, μ_t, at time t must ensure that the movie block requested by client C is available in at least one of the servers in $\mu_t(C)$. That is,

$$\texttt{data}(C, t) = (\mathcal{M}, s, \ell, step) \rightarrow \forall(0 \leq i \leq \ell)\exists(j \in \mu_t(C)(j \in \wp(\mathcal{M}, s + (i \times step))$$

Thus, for example, suppose a client is fast forwarding as shown in Figure 3. In this case, $\mathtt{data}(C,t) = (\mathcal{M}_j, 5, 5, 3)$. Suppose $\mu_t(C) = \{1,3,4\}$, i.e. client C is being serviced by servers $1, 3$, and 4, respectively. As described in Figure 3, client C's fast forward rate requires that blocks 5,8,11,14,17 be obtained for him. The placement constraint merely says that block 5 of the move $\mathcal{M}_j$ must be in one of servers $\{1,3,4\}$, block 8 must be in one of servers $\{1,3,4\}$, block 11 must be in one of servers $\{1,3,4\}$, and so on.

In addition to the placement constraint, an MOD system must satisfy several other constraints as well. Two important constraints are listed below: in the first, for each disk server, the bandwidth associated with that server should exceed the total bandwidth requirements of the clients assigned to that server. The second type of constraint specifies that the buffer available at any disk server should exceed the buffer requirements of the server at that time. It is particularly important to note that in contrast to several disk models, our framework keeps track of how things (e.g. consumption rates, cycle times, buffer requirements, etc.) change with time. This is critical in any realistic model of disk utilization and performance as these entities change with time and the demands placed by clients on the system.

Definition 2.6 (Disk Availability Constraint) A *disk availability constraint (DAC) about disk server i under placement mapping* $\wp$ at time t is a constraint of the form:

1. **Consumption Rate Constraint:** The sum of the consumption rates of the clients that are mapped to server i plus the total amount of bandwidth wasted during switchings between servers must be less than or equal to the bandwidth of the server i:

$$\left(\sum_{j \in \mathtt{d\_active}(i,t)} \mathtt{cons(j,t)} + \frac{\mathtt{switchtime(i,t)} \times \mathtt{dtr(i)}}{\mathtt{cyctime}(i,t)} \right) \leq \mathtt{dtr(i)}$$

   Note that if there is no single disk that can provide the consumption rate required by the client j, then two or more disks may be able to cooperate in order to match the required consumption rate. The **QuickSOL** algorithm that we propose later in this paper supports cooperative satisfaction of requests through a mechanism called *splitting* (of jobs).

2. **Buffer requirement constraint:** The sum of the buffer space requirements of the clients mapped to server i must be less than or equal to the available buffer space at server i.
   If $\mathtt{bufreq}(j,i,t)$ denotes the buffer space needed at server i to match the consumption rate of client j, then the above requirement can be formalized as follows:

$$\left(\sum_{j \in \mathtt{d\_active}(i,t)} \mathtt{bufreq(j,i,t)} \right) \leq \mathtt{buf(i)}$$

   In order to precisely compute $\mathtt{bufreq}(j,i,t)$, we observe that the buffer requirement of client j at server i is a function of the consumption rate of the client j, the bandwidth of server i, and the cycle time of server i:
   In order to match the consumption rate of client j, the server must ensure that the available cycle-time at server i is allocated in a manner proportional to the consumption rate of the client j, i.e. we must have

$$\mathtt{timealloc}(i,j,t) = \mathtt{cyctime}(i,t) \times \frac{\mathtt{cons}(j,t)}{\mathtt{dtr}(i)}.$$

As the data that is read within $\texttt{timealloc}(i, j, t)$ units of time must be kept in the buffers of server i, the number of buffers needed by client j is the difference of 1) the amount of data read by the disk in $\texttt{timealloc}(i, j, t)$ and 2) the data consumed in that time by client j.. This is given by:

$$\begin{aligned} \texttt{bufreq}(j, i, t) &= (\texttt{dtr}(i) - \texttt{cons}(j, t)) \times \texttt{timealloc}(i, j, t) \\ &= (\texttt{dtr}(i) - \texttt{cons}(j, t)) \times \frac{\texttt{cons}(j, t)}{\texttt{dtr}(i)} \times \texttt{cyctime}(i, t). \end{aligned}$$

$\diamond$

The above constraints specify the basic constraints that tie together, the resources of the MOD disk server system, and the requirements of the clients. Using these definitions, we are now ready to define the *state* of an MOD system.

2.3 State of an MOD System

In our framework, we assume that at any given point in time, the essential characteristics of an MOD system are captured through the notion of *state*. At any given point in time, the MOD system is in a state $\mathcal{S}(t)$ and certain *events* occur (e.g. new clients enter, old clients leave, some continuing clients "pause", others view the movie normally, while yet others may fast-forward or rewind). The aim of this paper is to show that these different events can be *neatly* handled by modifying the current state to a new state that satisfies the needs of the clients. Thus, the concept of "state" defined below is central to this paper.

Definition 2.7 (State of an MOD System) The *state*, $\mathcal{S}(t)$, of a movie-on-demand system at time t consists of the following components:

1. A set $\texttt{active}(t)$ of active clients at time t;
2. The current $\texttt{cyctime}(i, t)$ of each server in the system;
3. The consumption rates of the active clients ($\texttt{cons}(i, t)$) at time t;
4. The time, $\texttt{timealloc}(i, j, t)$, within $\texttt{cyctime}(i, t)$ that has been allocated by server i to client j at time t;
5. The information requirements ($\texttt{data}(i, t)$) of each active client at time t.
6. A client mapping μ_t at time t; $\diamond$

Note that given $\mathcal{S}(t)$, we can *automatically determine* what constraints are to be satisfied.

Definition 2.8 If $\mathcal{S}(t)$ is a state, then the *set,* $\texttt{CONSTR}(t)$, *of constraints* to be enforced at time t is:

1. the set $\texttt{DAC}(1, t), \ldots, \texttt{DAC}(N, t)$ of disk availability constraints where N is the number of servers;

2. The set $\texttt{PC}(1,t),\ldots,\texttt{PC}(M,t)$ of placement constraints where M is the number of *active* clients;

3. The set of CycTime and Allocation Constraints at time t. $\diamond$

However, a careful examination of the definition 2.6 will show that the disk availability contraints implicitly handle the CycTime and Allocation constraints as well. Hence, in our algorithms we do not explicitly check for the satisfiability of the cycletime and the time allocation constraints explicitly as their satisfaction is implied by satisfaction of the disk availability constraints.

A state $\mathcal{S}(t)$ is said to be **valid** iff it satisfies $\texttt{CONSTR}(t)$. Our algorithm works with *any* set of constraints. Consequently, if in the future, researchers were to develop more fine-grained models of disk characteristics, and if these led to new constraints, then all we would need to do is to "augment" the set $\texttt{CONSTR}(t)$ with these additional constraints and all the definitions in the paper and the algorithms would apply to such an extended framework as well.

Note that the total buffers available at servers ($\texttt{buf}(i)$), the total bandwidth of the servers ($\texttt{dtr}(i)$), the placement mapping ($\wp$), and context switch times of the servers ($\texttt{switchtime}(i,j,d)$) are assumed to be constant. If we wish to make any of these dynamic, they can easily be extended to have one extra parameter, viz. t, and can be explicitly included in the state definition). Hence they are not included in the state definition.

We have now defined the concept of a state of a movie-on-demand system formally. In the next section, we will define the concept of an *event*. Subsequently, we will show how events cause state-changes. In other words, if the MOD system is in state $\mathcal{S}(t)$ at time t, there may be several states, $\mathcal{S}_1(t+1),\ldots,\mathcal{S}_k(t+1)$ that "incorporate" the events – of these contenders, the MOD system must choose "the best" one according to some objective function.

3 Events

In any movie-on-demand system, at a given point t in time, one or more of a limited set of client-related events occur. Some of the possible events are listed below. Our model is capable of handling each and every one of these events by adjusting the values of the special $\texttt{data}(-,-)$ characteristics of clients.

(New Client Enters) This is a situation when a new client C_i enters the system and requests block 0 of a movie $\mathcal{M}$. This can be modeled by setting $\texttt{data}(i,t) = (\mathcal{M}, 0, \ell, 1)$.

(Old Client Exits) This is a situation when an existing client C_i leaves the system: this can be modeled by setting $\texttt{data}(i,t) = \perp$ meaning that the client is not accessing data any more.

(Continuing Clients) Continuing clients, C_i, may do one of the following things:

- **(Normal)** They request a new block of the movie at the normal playback speed, i.e. if $\texttt{data}(i,t-1) = (\mathcal{M}, s, \ell, k)$, then $\texttt{data}(i,t) = (\mathcal{M}, s+(\ell \times k), \ell, 1)$.
- **(Pause)** In this case, if $\texttt{data}(i,t-1) = (\mathcal{M}, s, \ell, k)$, then $\texttt{data}(i,t) = (\mathcal{M}, s+(\ell \times k), \ell, 0)$ indicating that the client is requesting only one block (s).

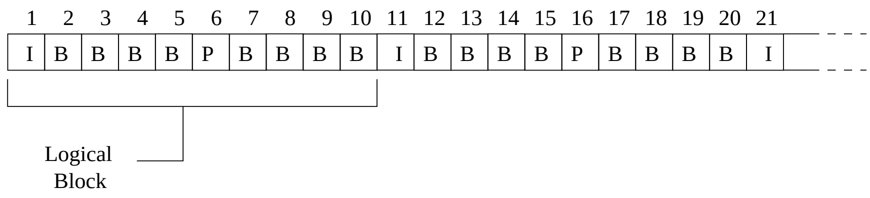

Figure 5: MPEG stream

- **(Fast Forward)** In this case, if $\texttt{data}(i, t-1) = (\mathcal{M}, s, \ell, k)$, then $\texttt{data}(i, t) = (\mathcal{M}, s + (\ell \times k), \ell, \texttt{ff})$ where ff is the fast-forward factor.
- **(Rewind)** In this case, if $\texttt{data}(i, t-1) = (\mathcal{M}, s, \ell, k)$, then $\texttt{data}(i, t) = (\mathcal{M}, s + (\ell \times k), \ell, -\texttt{rf})$ where rf is the rewind-factor.

Note that when ff (rf) is greater than 1, then the movie is displayed with a speed greater than the normal speed. However, in many cases, the speed-up that the user requests may not equal to the fast-forward factor (rewind-factor). The relationship between the fast-forward factor and the speed-up factor requested by the user depends on the storage format of the movie. For instance, the following example shows this relationship for movies stored in MPEG format:

Example 3.1 (MPEG Example) MPEG videos are stored as a sequence of I, P and B frames (figure 5). I frames are independent images, while B and P frames are calculated from the I frames using interpolation. The MPEG video sequence is often considered to be composed of logical blocks where each logical block has one I frame and various P and B frames (figure 5).

In such an environment, the fast-forward operation can be performed by displaying only the I frames and by skipping some of the logical blocks (figure 6).

The parameters that define an MPEG stream are the following:

- S_I: Size of an I frame.
- S_B: Size of a B frame.

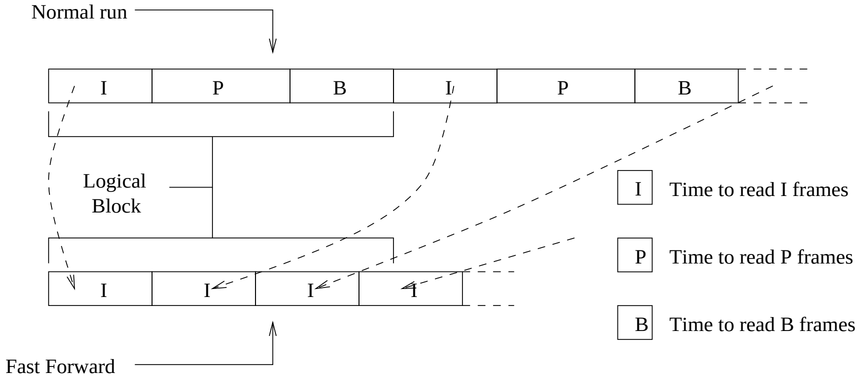

Figure 6: Comparison of the normal run and fast-forward operations in MPEG

- S_P: Size of a P frame.
- N_B: Number of B frames in a logical block.
- N_P: Number of P frames in a logical block.

Suppose we let ff denote the fast-forward ratio, S_L denote the size of a logical block, and N_L denote the number of frames in a logical block. The following equations enable us to compute the size of a logical block and the number of frames in a logical block:

$$S_L = S_I + N_B \times S_B + N_P \times S_P$$
$$N_L = 1 + N_B + N_P$$

If the system ignores the B and P frames, then it can read/transfer $(\frac{S_L}{S_I})$ I frames in the time required to read/transfer one logical block.

On the other hand, one I frame is worth $ff \times N_L$ frames, and one logical block is worth only N_L frames. Hence, during fast-forward, the system can consume $\frac{S_L}{S_I} \times ff \times N_L$ worth frames in time necessary to consume N_L frames. This leads to a speed-up factor $\mathcal{S}$ which can be calculated as follows:

$$\mathcal{S} = \frac{S_L}{S_I} \times ff$$

The above equation shows how we may derive the speed-up factor, given the fast-forward factor. One can also use this equation to calculate ff from the speed-up factor:

$$ff = \mathcal{S} \times \frac{S_I}{S_L} = \mathcal{S} \times \frac{S_I}{S_I + N_B \times S_B + N_P \times S_P}$$

□

We now provide a formal definition of events.

Definition 3.1 (Event) An *event* occurring at time t is a statement of the form $\texttt{data}(i,t) = (\mathcal{M}, s, \ell, k)$. Given a time instant t, we use $\texttt{EV}(t)$ to denote the *set* of events that occur at time t. For all t, we assume that $\texttt{EV}(t)$ cannot contain two or more items of the form $\texttt{data}(i,t) =< \cdots >$, i.e. a client is allowed to do at most one thing at a time (which seems to make intuitive sense, anyway).

As an example, consider the event set:

$$\texttt{EV}(5) \quad = \quad \{\texttt{data}(3,5) = (\mathcal{M}_1, 0, 30, 1); \texttt{data}(2,5) = \perp; \texttt{data}(1,5) = (\mathcal{M}_3, 45, 30, 1)\}.$$

The event $\texttt{data}(3,5) = (\mathcal{M}_1, 0, 3, 1)$ says that client C_3 is entering the system and wants to see movie $\mathcal{M}_1$. The reason we know this client is "new" to the system is because he is watching the movie from block 0. On the other hand, the event $\texttt{data}(2,5) = \perp$ indicates that client C_2 is exiting the system. Finally, $\texttt{data}(1,5) = (\mathcal{M}_3, 45, 30, 1)$ indicates that client C_1 is a continuing client who is watching the movie in "normal" mode.

It is easy to see that the occurrence of events should cause the state of the MOD system to change. In particular, events may require the allocation of servers to new clients in such a way that the servers can satisfy the clients' needs. Furthermore, events may require previous client mappings to be "re-assigned"; for example, at time 2, it may be the case that server 3 has a full load; however, an incoming client may require a movie that is *only* available on server 3; hence in this case, an existing client of server 3 must be switched (if possible) to some other server. In our architecture (cf. Figure 2), the job of the router is to find a new state (which includes a possibly new client mapping) that:

- satisfies all the current needs of the clients (if possible) and
- finds an optimal (to be defined in Definition 3.3 below) state that achieves this.

We now define these two concepts formally.

Definition 3.2 (Evaluation Function) An *evaluation function* is a function which takes as input, a state S and returns a numeric value. $\diamond$

In the following section, we present some evaluation functions that may be used in the design of a movie-on-demand system. Any of these metrics can be chosen within our framework because essentially, they all measure some aspect of a state. In general, it is more appropriate to use a combination of these metrics. We also show how these metrics can be put together in order to handle different aspects of efficiency.

As it is well known that unary functions can be used to model n-ary functions for arbitrary n by using a technique called *currying* (due to Haskell B. Curry), the above definition is very robust and allows MOD system managers to specify evaluation functions that take other parameters into account as well. For example, suppose a systems manager has some statistical data, `Stat` that he wants to be considered when evaluating a state. Then we may set $eval(S) = eval_{\texttt{Stats}}(S)$ where $eval_{\texttt{Stats}}$ looks at the body of statistical data the MOD system manager has. Use of histories may be similarly encoded.

Before introducing specific evaluation functions to improve MOD system efficiency, we first define what it means to (optimally) "handle" the occurrence of events.

Definition 3.3 (Optimal Event Handling) Suppose $\mathcal{S}(t)$ is a valid state at time t and $\texttt{EV}(t)$ is a set of events occurring at time t. State $\mathcal{S}_1(t+1)$ is said to *handle the events* $\texttt{EV}(t)$ iff the following conditions hold:

1. $\texttt{active}_{\mathcal{S}_1}(t+1) = (\texttt{active}_{\mathcal{S}}(t) - \{C \mid C$ is a client exiting at time t according to $\texttt{EV}(t)\}) \cup \{C \mid C$ is a client entering at time t according to $\texttt{EV}(t)\}$, **and**
2. $\mathcal{S}_1(t+1)$ satisfies all constraints in $\texttt{CONSTR}(t+1)$.

We say that state $\mathcal{S}_1(t+1)$ *optimally handles the events w.r.t. an evaluation function eval* $\texttt{EV}(t)$ iff

1. $\mathcal{S}_1(t+1)$ handles the events and

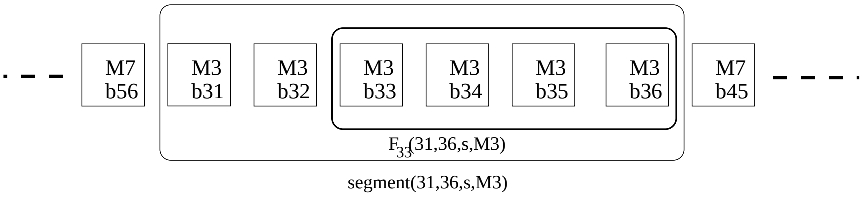

Figure 7: Forward span of $segment(31, 36, s, M3)$ at block 33

2. there no state, $\mathcal{S}_2(t+1)$ that handles the events such that $eval(\mathcal{S}_2(t+1)) > eval(\mathcal{S}_1(t+1))$.

The reader will note that according to this definition, when the system is in state $\mathcal{S}(t)$ and events $\texttt{EV}(t)$ occur, we are interested in finding a state $\mathcal{S}(t+1)$ such that $\mathcal{S}(t+1)$ satisfies the constraints in $\texttt{CONSTR}(t+1)$ *and* the evaluation of $\mathcal{S}(t+1)$ is as high as possible. There is no loss of generality in assuming that we are always solving "maximization problems." As is well-known in operations research, if z is an objective function and C is a set of constraints, the problem: "**minimize** z **subject to** C" is equivalent to the problem "**maximize** $-(z)$ **subject to** C".

3.1 Evaluation Functions

In this section, we provide a number of different metrics that improves the efficiency of a MOD system. These different metrics (or evaluation functions) reflect different intuitions on how to measure the performance of a multimedia heterogeneous disk server system. These evaluation functions will be defined formally in terms of some expressions described below:

Definition 3.4 (Forward Span) The *forward span of segment(i,j,s,m) at block k where* $i \leq k \leq j$ is: $F_k(i, j, s, m) = \{b \mid k \leq b \leq j\}$. $\diamond$

Figure 7 shows the forward span of $segment(31, 36, s, M3)$ at block 33. Intuitively, if the client asks for a block with long forward span, then the server is likely to serve the client for a long time. Hence, the use of long forward spans reduces the number of times the router may have to switch servers.

Definition 3.5 (Backward Span) The *backward span of segment(i,j,s,m) at block k where* $i \leq k \leq j$ is: $B_k(i, j, s, m) = \{b \mid i \leq b \leq k\}$. $\diamond$

Let SM_k denote the set of pairs $< m, i >$ where i denotes a block and m denotes a movie such that $\wp(m, i) = k$. In other words, SM_k is the set of all pairs $< m, i >$ such that the i'th block of the movie m is available through disk server k. Furthermore, let $D(m, i)$ denote the set of servers k such that $< m, i > \in SM_k$. The replication ratio of a disk server s may then be defined as follows:

Definition 3.6 (Replication Ratio) Let m and i denote a movie and a block, respectively. The *replication ratio,* R_s, of the disk server s is given by:

$$R_s = \frac{\sum_{(m,i) \in SM_s} |D(m, i)|}{|SM_s|}$$

Figure 8 provides an example of how replication ratios are calculated. Consider server A of Figure 8. Here, server A contains two movie blocks: $(m1, b22)$ and $(m2, b12)$. The total number of occurrences of $(m1, b22)$ in all three disk servers is 3 – one in each of disk servers A, B, C. However, the block $(m2, b12)$ occurs only in disk servers A, B. Thus, the numerator of the above expression is (3+2) and the denominator (i.e. number of blocks at server A) is 2; therefore, the replication ratio of server A is 2.5. The replication ratios of servers B and C may be similarly calculated.

Intuitively, if server S has a high replication ratio, and if S is unavailable for some reason (e.g. shortage of buffer space), then new clients have a good chance of accessing duplicate copies of blocks on server S by accessing other servers. However, if a server S has a low replication ratio, and if it is unavailable, then requests of new clients who desire relatively rare blocks may be delayed.

We are now in a position to define a suite of evaluation functions.

3.1.1 First evaluation function ($eval_1$): Maximize spans

At any given point in time, the active clients of a MOD system are either (1) reading future blocks (in the case of new clients, normal continuing clients, and clients who fast-forward) or (2) reading previous blocks ("rewind" clients), or (3) doing nothing (paused clients). A state which maximizes the forward-span of clients in category (1) and maximizes the backward spans of clients in category (2) will minimize server switching. The first evaluation function, $eval_1$, attempts to find the *total span* of the given state S by computing the sum of forward spans for clients in category (1) and the sum of backward spans for clients in category (2) and adding them. The following algorithm specifies this evaluation function.

$TotSpan(S) = 0$;
for all clients C *in the system do*;
    let $data(C, t)$ *be* $(M, start, len, step)$;
    if $step = 0$ *then* $TotSpan(S) = TotSpan(S)$;
    if $step > 0$ *then*;
        for all servers S *in* $\wp(M, start)$ *do*;
            Let there exists a $segment(i, j, S, M)$ *such that* $i \leq start \leq j$

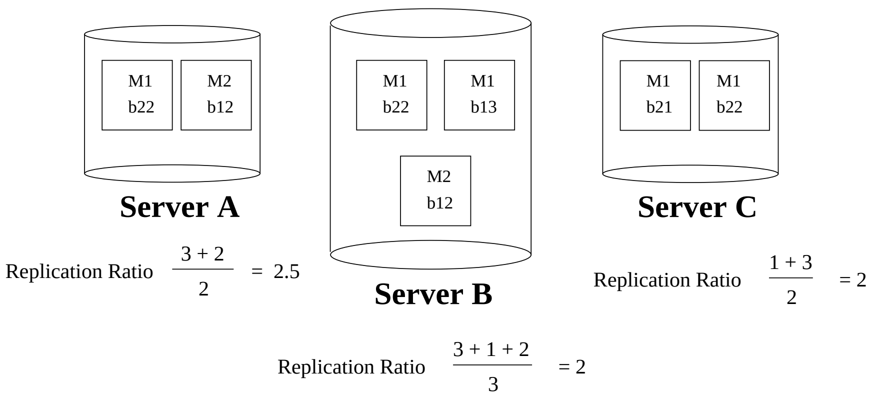

Figure 8: The replication ratios of servers A, B and C

$TotSpan(S) = TotSpan(S) + |F_{start}(i, j, S, M)|$;

if $step < 0$ *then*;

for all servers S *in* $\wp(M, start)$ *do*;

let there exists a $segment(i, j, S, M)$ *such that* $i \leq start \leq j$.

$TotSpan(S) = TotSpan(S) + |B_{start}(i, j, S, M)|$;

The evaluation function $eval_1(S_1)$ is defined as: $eval_1(S_1) = TotSpan(S_1)$.

3.1.2 Second evaluation function ($eval_2$): Use servers with high replication ratio

The second evaluation function ($eval_2$) tries to maximize the use of servers with high replication ratio. Recall that the replication ratio associated with server s is a measure of how much the data on that server is available on other servers. By assigning, to a client, a server s that has a high replication ratio, there is a good chance that other future clients may find the data on s present on other servers. To encourage assignment of clients to a server s with high replication ratios, we can set:

$$TotRep(S) = \sum_{s} ((|\texttt{d\_active}(s,t)|) \times R_s)$$

The evaluation function $eval_2$ is defined as: $eval_2(S_1) = TotRep(S_1)$

Example 3.2 As an example, suppose we have a system with three active clients, C_1, C_2, C_3 and suppose we have the three servers shown in Figure 8. Furthermore, suppose (in some state S) that server A serves clients C_1, C_2 and server B processes C_1 and server C processes all three. Then

$$TotRep(S) = (2 \times 2.5) + (1 \times 2) + (3 \times 2) = 13.$$

3.1.3 Third evaluation function ($eval_3$): Minimize load variance

The third evaluation function ($eval_3$) tries to maximize the load balance in the client-server mappings. Suppose $AvgClients(S)$ denote the average number of clients per server in the system in state S. If we return to Example 3.2, we observe that $AvgClients(S) = 2$.

Let $TotVariance(S)$ denote the total variance in the number of clients assigned to the servers in state S, i.e.

$$TotVariance(S) = \sum_{server\ s} (|\texttt{d\_active}(s,t)| - AvgClients(S))^2.$$

If we return to Example 3.2, we observe that $TotVariance(S) = (2-2)^2 + (1-2)^2 + (3-2)^2 = 2$.

Clearly, we would like to *minimize* total variance. As our framework always assumes that evaluation functions must be maximized, we can set $eval_3(S)$ to be equal to the minus of $TotVariance(S)$, i.e. $eval_3(S_1) = -TotVariance(S_1)$

3.1.4 Fourth evaluation function ($eval_4$): Minimize the number of switchings and splittings

The fourth evaluation function ($eval_4$) tries to minimize the cost of moving from the current state to a potential new state. The cost of going from one state to another depends fundamentally on the number of server switchings and clients splittings done. Hence we may wish to define an evaluation function that minimizes the number of servers switched and the number pf clients splitted. Thus we may define $eval_4$ as: $eval_4(S) = -($# *of server switchings and client splittings involved in going from current state to state* S)

3.1.5 Fifth evaluation function ($eval_5$): Maximize the number of satisfied clients

In many cases, the load on a MOD system may be so high that not all client requests can be satisfied. In such cases, a simple evaluation metric would simply count the number of clients satisfied by a given state. Hence we can define an evaluation function which maximizes this number:

$$
\begin{aligned}
eval_5(S_1) &= (\#\ \textit{of satisfied clients}) \\
&= |\{i | \mathtt{data}(i,t)\ \textit{that is satisfied in state}\ (S_1)\}|
\end{aligned}
$$

3.1.6 Hybrid evaluation function ($eval_H$)

One of the great advantages of our framework is that the manager of an MOD system may use different evaluation functions to assess the performance of his system.

For example, suppose the MOD systems manager wishes to create a hybrid evaluation function using the functions $eval_1, \ldots, eval_5$ described earlier. He could assign numbers $\beta_1, \ldots, \beta_5$ assessing the respective relative importance of these functions – thus, if β_1 was 1 and β_2 was 3, then this means that the system manager is saying the evaluation criterion $eval_2$ is three times more important than $eval_1$ for his needs. In general, the MOD system manager could set the β_i's using a menu, and the system would generate the evaluation function:

$$
\begin{aligned}
eval_H(S_1) = &(\beta_1 \times eval_1(S_1)) + (\beta_2 \times eval_2(S_1)) + (\beta_3 \times eval_3(S_1)) + \\
&(\beta_4 \times eval_4(S_1)) + (\beta_5 \times eval_5(S_1))
\end{aligned}
$$

4 Computing State Transitions: The QuickSOL Algorithm

In the preceding section, we have given a formal, declarative definition of how external events in media systems cause a change in the system state and furthermore, we have specified what constitutes an optimal new state that handles the events. The purpose of this section is to develop an efficient way of processing events. In particular, if the MOD system is in state $\mathcal{S}(t)$ and the events in $\mathtt{EV}(t)$ occur, we would like to find an optimal state, $\mathcal{S}(t+1)$, that handles the event. However, as in reality, there are often real-time constraints, we would like the computation of such optimal states "interruptible"

– in other words, the algorithm should quickly compute some state that handles the events in `EV`(t). This state can then be optimized further till either the deadline is reached or till an optimal state is computed, whichever is earlier. Our algorithm, called **QuickSOL** (cf. Section 4.2, does exactly this. There are two phases to the **QuickSOL** algorithm – the first phase, called `FindSOL` computes a quick (often non-optimal) solution that accommodates the clients' needs (if possible). The second phase, called `OptimizeSOL` ameliorates the solution found in the `FindSOL` phase. Later, in Section 5, we will report on various experiments that evaluate the **QuickSOL** algorithm.

Before proceeding to describe the **QuickSOL** algorithm in detail, we first present some structures used by it.

4.1 State and Resource Tables

The MOD system keeps the information necessary for running the **QuickSOL** algorithm, i.e. the state of the MOD system and the resources available at the servers, in tables. In section 2.3 we introduced the pieces of information necessary to identify a state of the MOD system. Similarly, in section 2.2 we listed the types of resources that a video server must have.

Hence, the **QuickSOL** algorithm uses two tables a *state table* and a *resource table* to find a client mapping which satisfies the input events.

The state table contains fields decribing the state of a MOD system at time t. Some of these fields are: $Client$, $Server$, $Movie$, $Start$, $Length$,, and $Step$. Note that a single client may be served by multiple servers at a given point in time, and this may be represented in the state table as multiple tuples involving the same client. Similarly, the resource table contains fields such as $Server$, $AvBuf$, $AvCycTime$, and $AvBWidth$ which describe the resources available at the servers at time t. Clearly, only one tuple is needed per server in the resource table.

4.2 The QuickSOL Algorithm: `FindSOL` Phase

The nature of multimedia data requires that the system be able to quickly and efficiently respond to clients' needs. In particular, the system should be able to handle new requests by smoothly switching clients from one server to another if needed. This task is particularly important when a server cannot satisfy the needs of a client. If for instance, the movie segment requested by the client is not available in the server it is currently assigned to, then the client may be switched to an other server which has these segments.

The **QuickSOL** algorithm first processes clients exiting the system – this is done right at the beginning as such clients free up resources that may be allocated to other clients. The status of the table is unchanged by "Paused" clients.

The **QuickSOL** algorithm works by assigning each event a *priority*. All non-paused continuing clients are initially assigned the highest priority (viz. 4) while all new clients are assigned priority 2. This way, clients that are already in the system are serviced before the clients that are waiting to get in.

These priorities get "adjusted" as the algorithm computes. When processing a non-paused continuing client C, we first try to see if the data being requested by C is available on the server(s) that C is currently assigned. If not, C must be "switched" to another server that has the desired data. We then re-set C's priority to 4 and consider it as a "New" client with priority 4 (i.e. this client has higher priority than other "new" clients who had priority 2). This prioritization ensures that continuing clients receive "higher" priority than new clients (who have a priority of 2 in general).

Processing new clients (and splitting events): We process new clients C as follows: we first identify the servers that have the data required by C. We then determine which of these servers have enough available bandwidth to satisfy the consumption rate of C'.

If no such servers are available, we *split* the event involving client C into two "sub-events." For instance, if client C is involved in the event $\mathtt{data}(C,t) = (\mathcal{M}, s, \ell, step)$, then we can split this into two sub-events, $\mathtt{data}(C,t) = (\mathcal{M}, s, \lfloor \ell/2 \rfloor, 2 \times step)$ and $\mathtt{data}(C,t) = (\mathcal{M}, s + step, \lceil \ell/2 \rceil, 2 \times step)$. One can think of this as a form of striping[21, 18] – however, there are significant differences also. In striping, *data* is striped across disks to effect an increase bandwidth. In contrast, in event splitting, a client request (i.e. event) is broken up into smaller events that may be satisfied by distributing the load across multiple servers. As each of these two sub-events have "smaller" requirements than the event they were derived from, it may be possible to find two different servers that can jointly satisfy this request. Thus, these two sub-events are assigned priority 4 and added to the "new" list. It is important to note that *both* these sub-events must be accomplished for the event from which they are derived to be accomplished. The same process is repeated when we consider buffer requirements of the clients instead of consumption requirements.

Once all this has been completed, we are left with the following: for each new event e_i of the form $\mathtt{data}(cl_i, t) = (\mathcal{M}, s, \ell, step)$, we have a set $FIN(e_i)$ specifying which servers satisfy all the requirements of the client/subclient involved in event e_i. If $C_1, \ldots, C_r$ are all the active clients at time $(t+1)$, we must find sets $st_1, \ldots, st_r$ such that each $st_i \subseteq FIN(i)$ and if client C_i is involved in many (sub)events, then st_i must contain at least one member in $FIN(C_i)$.

Linked Events: We say that two events $e_1 : \mathtt{data}(C_1, t_1) =< quadruple >$ and $e_2 : \mathtt{data}(C_2, t_2) =< quadruple >$ are *linked* iff $C_1 = C_2$ and $t_1 = t_2$. If two events are linked, then this means that these are two sub-events generated by the splitting of a single client.

FindSOL algorithm:

1. Split $\mathtt{EV}(t)$ into 6 sets: $\mathtt{new}(t)$, $\mathtt{exit}(t)$, $\mathtt{cont}(t)$, $\mathtt{pause}(t)$, $\mathtt{ff}(t)$ and $\mathtt{rew}(t)$ depending upon whether the client involved in the event is a new client, exiting client, normal-continuing client, continuing client doing a pause, continuing client doing a fast-forward, continuing client doing a rewind, respectively. For each event in $\mathtt{new}(t)$ there is a *priority* $(0 \ldots 4)$.
2. Set the priority of all events in $\mathtt{new}(t)$ to 2.
3. For all events e_i set $link(e_i)$ to $\emptyset$. (No client has been split yet).
4. (Handle Exiting Clients) For each Client C_i in $\mathtt{exit}(t)$ do:
   (a) Let $r_1, \ldots, r_w$ be all rows in the state table corresponding to client C_i.
   (b) For $j = 1, \ldots, w$ do:

- If r_j.`Server` = k, then update the record in the Resource Table corresponding to server k appropriately.
- Delete row r_i from the state table.

5. (Handling Continuing-Pausing Clients) For each Client C_i in `pause`(t) do **nothing**.
6. (Handling Continuing Clients) For each event e_i in `cont`(t) $\cup$ `ff`(t) $\cup$ `rew`(t) where e_i is of the form `data`$(C_i, t) = (\mathcal{M}_i, s_i, \ell_i, step_i)$ do:

   (a) Let $r_1, \ldots, r_w$ be all rows in the state table with `Client =` C_i.

   (b) **If** for all $j = 1, \ldots, w$ r_j.`Server` satisfies the constraints, **then**

      i. Modify each $r_j.Start$, $r_j.Length$, and $r_j.Step$ to match the new $start_i$, $length_i$ and $step_i$.

   else

      i. Move event e_i into `new`(t) with the priority set to 4.
      ii. Update the Resource Table as in Step 4b.
      iii. Delete each r_j from the state table.

7. (Handling New Clients) Let $e_1 \ldots e_m$ be all the events in `new`(t), and let `nlist`(t) be the list of events in `new`(t) sorted in descending order of priority.

   (a) While `nlist`$(t) \neq \emptyset$ do

      i. Let e_i be $head$(`nlist`(t)) where e_i is of the form of $data(C_i, t) = (\mathcal{M}, s, l, step)$). Set `nlist`$(t)$ to $tail$(`nlist`(t)).
      ii. Let $SRV(e_i) = \{s \mid s \text{ is a server that satisfies the placement constraint of movie } \mathcal{M}\}$
      iii. If $SRV(e_i) = \emptyset$, then (* split event e_i into smaller events *)

         A. If the length l smaller than $MINLENGTH$, then mark the event e_i as unsatisfiable, delete all the events in $link(e_i)$ from `new`(t), and undo the effects of any events in $link(e_i)$ that have already been processed. Then go to 7a. **else..**
         B. If the priority of e_i is 4, then set the priority of e_i to 3 and put it back into `nlist`(t) and go to 7a
         C. If the priority of e_i is 2, then set the priority of e_i to 1 and put it back into `nlist`(t) and go to 7a
         D. If the priority of e_i is neither 2 nor 4, then
            - Let e'_i be $data(C_i, t) = (\mathcal{M}, s, \lfloor l/2 \rfloor, step))$, and $link(e'_i) = link(e_i) \cup e''_i$.
            - Let e''_i be $data(C_i, t) = (\mathcal{M}, s + \lfloor l/2 \rfloor, \lceil l/2 \rceil, step))$, and $link(e''_i) = link(e_i) \cup e'_i$.
            - Place e'_i and e''_i at the head of `nlist`, and go to 7a.

      iv. Otherwise ($SRV(e_i) \neq \emptyset$): Let $CONS(e_i) \subseteq SRV(e_i)$ be the set of servers that can satisfy the consumption rate of client C_i.
      v. if $CONS(e_i) = \emptyset$ then (* split event e_i into smaller events *)

         A. If the length l smaller than $MINLENGTH$, then mark the event e_i as unsatisfiable, delete all the events in $link(e_i)$ from `new`(t), and undo the effects of any events in $link(e_i)$ that have already been processed. Then go to 7a. **else..**
         B. If the priority of e_i is 4, then set the priority of e_i to 3 and put it back into `nlist`(t) and go to 7a
         C. If the priority of e_i is 2, then set the priority of e_i to 1 and put it back into `nlist`(t) and go to 7a
         D. If the priority of e_i is neither 2 nor 4, then
            - Let e'_i be $data(C_i, t) = (\mathcal{M}, s, \lfloor l/2 \rfloor, step * 2))$, and $link(e'_i) = link(e_i) \cup e''_i$.
            - Let e''_i be $data(C_i, t) = (\mathcal{M}, s + step, \lceil l/2 \rceil, step * 2))$, and $link(e''_i) = link(e_i) \cup e'_i$.

- Place e_i' and e_i'' at the head of `nlist`, and go to 7a.

vi. Let $BUF(e_i) \subseteq CONS(e_i)$ be the set of servers that can satisfy the buffer requirement of client C_i.

vii. if $BUF(e_i) = \emptyset$ then (* split event e_i into smaller events *)

A. If the length l smaller than $MINLENGTH$, then mark the event e_i as unsatisfiable, delete all the events in $link(e_i)$ from `new`(t), and undo the effects of any events in $link(e_i)$ that have already been processed. Then go to 7a. **else..**

B. If the priority of e_i is 4, then set the priority of e_i to 3 and put it back into `nlist`(t) and go to 7a

C. If the priority of e_i is 2, then set the priority of e_i to 1 and put it back into `nlist`(t) and go to 7a

D. If the priority of e_i is neither 2 nor 4, then

- Let e_i' be $data(C_i, t) = (\mathcal{M}, s, \lfloor l/2 \rfloor, step * 2))$, and $link(e_i') = link(e_i) \cup e_i''$.
- Let e_i'' be $data(C_i, t) = (\mathcal{M}, s + step, \lceil l/2 \rceil, step * 2))$, and $link(e_i'') = link(e_i) \cup e_i'$.
- Place e_i' and e_i'' at the head of `nlist`, and go to 7a.

viii. Otherwise (i.e. $BUF(e_i) \neq \emptyset$), if s_j is the member of $BUF(e_i)$ which returns the maximum evaluation value, and if e_i is of the form $data(C_i, t) = (\mathcal{M}, s, l, step)$, then add $C_i, s_j, \mathcal{M}, s, l, step)$ to the state table, and update the Resource table accordingly.

As pointed out earlier in the paper, the state computed by the `FindSOL` algorithm is *not* guaranteed to be optimal. The reason for this is that real-time requirements dictate that some solution be computed very fast, even if the solution so obtained is not optimal. The next subsection presents a way to improve on the solution returned by the `FindSOL` algorithm.

The worstcase running time of the `FindSOL` algorithm is

$$\begin{aligned} &O(num_{pause} + \\ &\quad len \times (num_{exit} + num_{cont} + num_{ff} + num_{rew}) + \\ &\quad len^2 \times (num_{new} + num_{cont} + num_{ff} + num_{rew})), \end{aligned}$$

where num_{pause}, num_{exit}, num_{cont}, num_{ff}, num_{rew}, and num_{new} denote the number of events in the corresponding sets, and len denotes the maximum sized movie piece requested by the events in `EV`(t):

$$len = max\{L | data(C_i, t) = (M, S, L, Step)\, is\, in\, ev(t)\}$$

In fact, len is the maximum number of the split events that may result from a single event e, and len^2 is the maximum number of splits that may be tried before satisfying event e.

Since the `FindSOL` algorithm gives precenedence to the continuing events, and since it does not perform event switchings unless some of the constraints are not satisfied, the algorithm is not necessarily complete. Hence, we need a second phase which to satisfy all possible clients, and to optimize the solution returned by the `FindSOL` algorithm.

4.3 QuickSOL Algorithm: `OptimizeSOL` Phase

Suppose $\mathcal{S}_0(t+1)$ is the state returned by the **QuickSOL** algorithm's `FindSOL` phase. The `OptimizeSOL` Phase of the **QuickSOL** algorithm tries to do three things to $\mathcal{S}_0(t+1)$:

1. **Switching:** It tries to switch clients from one server to another.
2. **Splitting:** It may split an event – for example, if, $\texttt{data}(C_i, t+1) = (\mathcal{M}, s, \ell, step)$ is one of the data requirements in state $\mathcal{S}_0(t+1)$, then it might split this into two "linked" clients.
3. **Inserting:** It tries to insert the so far *unsatisfied* events to the system.

The *optimization search space (OSS)* generated by the state $\mathcal{S}_0(t+1)$ is the set of all states, $\{s \mid s$ is a state that handles the events in $\texttt{EV}(t)$ and $eval(s) \geq eval(\mathcal{S}_0(t+1))$ and s is the state obtained by performing a sequence of switching/splitting operations starting from state $\mathcal{S}_0(t+1)$.

The `OptimizeSOL` phase of the **QuickSOL** algorithm does the following: initially, `bestSolution` is set to $\mathcal{S}_0(t+1)$. Whenever the algorithm computes a new state s, we set `bestSolution` to whichever of the two states s and the current `bestSolution` have a higher evaluation according to the evaluation function being used.

Note that, in order to swap two clients each assigned to a different server, we need (at least conceptually) a "working" memory area/space to temporarily store information pertaining to one of the clients. In order to accomplish this, we introduce a "virtual" (dummy) server V with infinite capacity. Note that, a client assigned to V is considered to be "unsatisfied", because V is only a temporary working space to perform client swapping. V cannot by itself service any requests. The only difference between a client assigned to V and a client which is not assigned to a server is that the former is assumed to be a client that has been accepted for service by the system and must be reassigned back to a normal server, whereas the later can be denied service if the current load on the system is too high and the resources needed by the client are unavailable.

OptimizeSOL algorithm:

1. Set `bestSolution` to $\perp$.
2. Set `bestState` to $\mathcal{S}_0(t+1)$.
3. Let $NewStates = \{\mathcal{S}_0(t+1)\}$.
4. Let $OldStates = \emptyset$.
5. While $NewStates \neq \emptyset$ and time is not up do
   (a) Pick a state S from $NewStates$.
   (b) Remove S from $NewStates$ and put it into $OldStates$
   (c) Create all possible states ($NextStates$) ,i.e. states that satisfy the constraints, from S by
      - switching events from one server to the other,
      - splitting events,
      - and inserting the so far *unsatisfied* events to the system.
   (d) Let $NextStates$ be $NextStates - (OldStates \bigcup NewStates)$.
   (e) Evaluate all states in $NextStates$.
   (f) If there are states in $NewStates$ with values higher than the value of the `bestState`, then pick the state (HS) with the highest value and let
      - `bestState` $= HS$.
      - If `bestState` handles all the input events than `bestSolution` = `bestState`

(g) Let $NewStates = NewStates \bigcup NextStates$

It is important to note that in Step 5 above, some events may be temporarily switched from a "real" server to the virtual server V meantioned earlier. We reiterate that events switched to the virtual server are considered "unsatisfied."

The **QuickSOL** algorithm is defined by first executing the `FindSOL` phase, and then executing the `OptimizeSOL` phase as described above. If there is a state that can handle $\texttt{EV}(t)$, then `bestSolution` describes a state which optimally handles the input events. On the other hand, if there is no state that can handle $\texttt{EV}(t)$, then `bestSolution` is equal to $\perp$. Yet, in that case `bestState` describes the best state found during the execution, and it can be used with the least penalty for the system.

Theorem 4.1 Suppose $\texttt{EV}(t)$ is the set of events that occur at time t and suppose $\mathcal{S}(t)$ was the state of the MOD system at time t. Let MINLENGTH= 1 in the **FindSOL** algorithm. Then:

- **(Soundness)** If $\mathcal{S}$ is a state returned by the **QuickSOL**, then $\mathcal{S}$ (successfully) handles the events in $\texttt{EV}(t)$.
- **(Completeness)** Suppose the set of states that can handle $\texttt{EV}(t)$ is non-empty. If the algorithm is not interrupted, then it is guaranteed to return a state that handles the events in $\texttt{EV}(t)$.
- **(Optimality)** Suppose the set of states that can handle $\texttt{EV}(t)$ is non-empty. If the algorithm is not interrupted, then it is guaranteed to return an optimal state (`bestSolution`) that handles the events in $\texttt{EV}(t)$.

Proof 4.1 The proof of theorem 4.1 is as follows:

- **(Soundness)** In order to show that the state $\mathcal{S}$ returned by the **QuickSOL** algorithm handles all the events in $\texttt{EV}(t)$, we need to show two things:

  1. That every record in the State Table after execution of **QuickSOL** satisfies all the constraints described in Definition 2.8.
  2. That every event in $\texttt{EV}(t)$ is handled by the State table on completion of the **QuickSOL** algorithm.

  We first observe that the **OptimizeSOL** algorithm ensures that any state it outputs satisfies all the constraints by virtue of Step 5c of that algorithm; hence, we only need to ensure that the manipulations performed by the **FindSOL** algorithm satisfy these same constraints.

  PART 1: Suppose the handling of an event e of the form $data(Client, t) = (Movie, Start, Length, Step)$ is assigned to the server $Server$ in the state table $\mathcal{S}(t+1)$ after the **QuickSOL** algorithm completed its execution. If the event e was a continuing or paused event, and e did not need to be switched from $Server_{old}$ in $\mathcal{S}(t)$ to $Server$ in $\mathcal{S}(t+1)$ (that is it satisfied the placement constraint), then clearly $Server$ can serve the event e. On the other hand, suppose that event e is not a new client, or that e had to be switched to $server$ then it must have been placed in the State Table as a consequence of executing the **QuickSOL** algorithm. Hence, for the rest of the discussion we assume that e is a new client, or $Server_{old}$ could not satisfy the constraints.

The *split rank*, $\texttt{SR}(e)$, of event e is defined to be 0 if event e was explicitly present in $\texttt{EV}(t)$. Otherwise, $\texttt{SR}(e)$ is the number of splits that were made to an event in $\texttt{EV}(t)$ to obtain e. We proceed by induction on $\texttt{SR}(e)$.

Base Case: If $\texttt{SR} = 0$, then e was in $\texttt{EV}(t)$ and was never split. e cannot be an event involving an exiting client as it would not then be in $\mathcal{S}$ (cf. Step 4 of the **FindSOL** algorithm). Thus, e is either associated with a continuing client or a new client.

If e is associated with a continuing client, then (as its split rank is 0), either Step 6(b) succeeded, which means that `Server` can indeed satisfy the request in e, or it means that `Server` is in the set, $SRV(e)$, identified in Step 7(a)(ii) and that `Server` is also in $CONS(e)$ in Step 7(a)(iv) and $BUF(e)$ in Step 7(a)(vi) which means that `Server` has all the resources needed to satisfy this event.

The same reasoning applies if e is associated with a continuing client, the only difference is that step 6(b) never applies, only the steps above that arisein step 7.

Inductive Case: On the other hand, suppose $\texttt{SR}(e) = n + 1$. Then this means that event e was obtained by repeatedly splitting some event $e' \in \texttt{EV}(t)$ $(n + 1)$ times. Let this sequence of splits be denoted by

$$e' = e_0 \xrightarrow{\text{split}} e_1 \xrightarrow{\text{split}} e_2 \xrightarrow{\text{split}} \cdots \xrightarrow{\text{split}} e_{n+1} = e.$$

Clearly, for each $1 \leq i \leq n$, either $SRV(e_i)$ or $CONS(e_i)$ or $BUF(e_i)$ was empty, leading to a further split. As e_{i+1} was never split, this means that $BUF(e_{n+1})$ was nonempty and contained `Server` in it.

PART II: Every event in $\texttt{EV}(t)$ is in one of the six sets, $\texttt{new}(t)$, $\texttt{exit}(t)$, $\texttt{cont}(t)$, $\texttt{pause}(t)$, $\texttt{ff}(t)$ and $\texttt{rew}(t)$. Each of these sets is considered in turn in the **FindSOL** algorithm. It is easy to see that each member of $\texttt{exit}(t)$ is appropriately handled in Step 4 of the **FindSOL** algorithm, and each member of $\texttt{pause}(t)$ is appropriately handled in Step 5 of the **FindSOL** algorithm.

The sets $\texttt{new}(t)$, $\texttt{cont}(t)$, $\texttt{ff}(t)$ and $\texttt{rew}(t)$ are slightly more complicated. Each event in any of the sets $\texttt{new}(t)$ is handled in Step 7, while each event in one of the sets $\texttt{cont}(t)$, $\texttt{ff}(t)$ and $\texttt{rew}(t)$ are handled in Steps 6 and 7. Whenever a split occurs in steps 7(a)(iii), 7(a)(v) or 7(a)(vii), *both* the split sub-events are reinserted into the appropriate list. This ensures that at the end of the **FindSOL** algorithm, all events are properly handled, except for those that are marked as "unsatisfiable" in Steps 7(a)(iii)A, or 7(a)(v)A, or 7(a)(vii)A. As we know that the **QuickSOL** algorithm has halted with success, these must have been satisfied in the **OptimizeSOL** phase which guarantees satisfaction of constraints in Step 5(f). This completes the proof.

- **(Completeness)** If the states that can handle $\texttt{EV}(t)$ is non-empty, then there exists a non-empty set $(\mathcal{M})$ of client mappings μ that handles the set of input events. In order to prove that the **QuickSOL** algorithm is complete we have to show that if $\mathcal{M}$ is non-empty than it returns a client mapping in $\mathcal{M}$.

  By the Soundness Proof, we already know that if `bestState` is returned by the **QuickSOL** algorithm, then `bestState` handles all the events in $\texttt{EV}(t)$ – so we merely show that **QuickSOL** returns a client mapping if $\mathcal{M}$ is non-empty.

  If there is a client mapping μ that handles the set of input events, then that client mapping can be found by switching all of the events into the virtual server S, splitting them until every event reaches to their minimal sizes, and then switching the splitted events back to the corresponding servers in μ. Since, the **OptimizeSOL** checks all possible "next states" in Step 5(c), it will eventually discover this solution, and will return it as its output.

Note that it does not mean that **QuickSOL** will do all this to find the result. The above argument only shows that if there is a solution, then **QuickSOL** will find it.

- **(Optimality)** If the set of states that can handle $\mathtt{EV}(t)$ is non-empty, then by completeness, the returned state satisfies all constraints (it handles the input events). Furthermore, the **QuickSOL** algorithm checks all the possible states, and it returns the state with the maximum evaluation value it finds during its execution. Hence, there cannot be any other state that handles the input states, and that has a higher evaluation value. □

The **OptimizeSOL** algorithm has one disadvantage, viz. that the set *NextStates* may be very large. This algorithm encodes a procedure to search a search tree using the "best-first" search strategy studied in artificial intelligence [16]. One may think of **OptimizeSOL** as constructing a search tree where each node is a state, and *NextStates* represent the children of any given node by the application of Step 5(c) of the **OptimizeSOL** algorithm. Instead of expanding the node S "completely" (as in Step 5(c)), it is possible to apply various heuristics to expand it partially. For example, a simple heuristic to cut down on the width of this tree is to split events (relatively cheap) much more often than switching servers (more expensive). Such a strategy could use a "switch to split" ratio to decide (for example) that every 25 splits should have a corresponding switch. A more detailed approach would always attempt to split *unsatisfiable* events first and switch/split servers in accordance with the above ratio till a given unsatisfiable event it satisfied. One then could continue the process with the next unsatisfiable event until no more unsatisfied events remain.

5 Experiments

We have developed an experimental version of the **QuickSOL** algorithm described earlier. The experimental version uses the following assumptions: (1) cycle-time: 2 secs (as in [15]), (2) buffer size: average buffer size is 4.3 MB, (3) disk rate: 1.6 MB/sec, (4) number of movies: 40, each with 35000 blocks where a block is one frame, (5) segment size per disk: 6000 blocks (average), (6) total amount of time: 2000 cycles, (7) number of servers: 350, (8) user arrival rate: between 2-6 new clients per second (compare with Rotem and Zhao [19] who consider 1-2 viewer per second when 350 servers are used), (8) user consumption rate: 200 KB/s.

In the experiments outlined in this paper, we started a system "from scratch" (i.e. no clients) and injected clients into the system over a period of 2000 cycles (approx. 375 minutes). Clients were injected at different rates (ranging from 2-6 new clients per second) and under different assumptions about the number of disks per server and the amount of buffer size available through the server. When a disk server has n disks in it, we will assume that the bandwidth of the server is $n \times 1.6$ MB/s – this assumes that data has been striped across the disks accessed by this server. This is a reasonable assumption [2].

As more and more clients enter the system, the system becomes "less responsive." Appendix A contains 10 figures showing how `FindSOL` behaves during the last 200 cycles of this 4000 cycle run – in these cycles, the system is more or less "maximally" loaded. Each figure has a number of graphs, and these graphs reflect assumptions on the number of (striped) disks per server (ranging from 1 to

4) and the average buffer size (ranging from 1×4.3 MB to 4×4.3 MB). The graphs themselves are of two types –

1. those that plot the number of continuing, successfully satisfied clients vs. the arrival rate of clients and
2. those that plot the running time of `FindSOL` vs. the arrival rate.

Based on the figures in Appendix A, we may make the following assertions:

1. **Finding 1:** If we examine any of the figures that plot continuing clients supported vs. arrival rate, then we notice that there is not much change when we increase either just the buffer availability or just the amount of disk space. For example, consider figure C/A-4. If we consider the bottom row of graphs in this figure, we see little improvement despite the increase in available disk space. The reason for this is that the limiting factor is not the disk space, but rather the buffer space, and hence increasing the disk space alone is not very helpful. The same observation applies to the first column of graphs in this figure. However, simultaneous increase of both resources leads to a substantial improvement in the number of clients that can be supported.

   In the case of running time of `FindSOL`, we observe (cf. bottom row of Figure T/A-4, for instance) that an increase in just one of the resources alone without due consideration of the other resources involved, may merely lead to additional splitting, and hence, increased running time.

   In general, however, given an evaluation function (e.g. the load balancing strategy of Figures T/A-4, C/A-4), if we wish to successfully support n continuing clients at arrival rate a and within a time interval t,we may use the graphs (e.g. Figures T/A-4, C/A-4) as follows: For each graph in these figures, draw a vertical line through `arrival rate = a`. At the place where it hits the curve, draw a horizontal line to determine the number of continuing clients supported (in the case of cont.client vs. arrival rate graphs) and time requirement (in the case of response time vs. arrival rate graphs). Pick the graph with the least resource requirements where these horizontal projections have better performance than the desired response time and the desired number of continuing clients.

   For example, suppose we wish to find (from Figures T/A-4, C/A-4), the buffer/disk requirements to support 4000 continuing clients arriving at the rate of 2 clients/second, with a response time of under 1000 milliseconds. All graphs in Figure C/A-4 other than those in the bottom row and leftmost column satisfy that 4000 continuing client requirement at the given arrival rate. Thus, to satisfy the continuing client requirement we need at least 2 disks/server, and a buffer average of 2×4.3 MB. Any configuration with either 2 disks/server of 2×4.3 MB buffer are unable to satisfy the desired response time requirement (cf. Figure T/A-4). Thus, we need at least 3 striped disks/server and 3×4.3 MB buffer to solve our problem.

   The same principle can be applied to other situations where different performance requirements and/or different evaluation functions are used.

2. **Finding 2:** If we look at the number of continuing clients satisfied and the response time generated by using different evaluation functions, we find that the Evaluation Function 1 (which maximizes Span) leads to performance gains over the other evaluation functions *when the load is high*; however, when the load is low, the other evaluation functions yield comparable results.

Thus, in general, should a media on demand vendor wish to provide VOD services where high loads may occur (e.g. at peak viewing times) then we recommend that evaluation function 1 which maximizes span be used.

3. **Finding 3:** Under high load conditions, increased arrival rate does not really increase the number of continuing clients that can be supported by the system, except for one anomalous case (Figure T/A-4, row 1, column 4, arrival rate = 2, where the execution time decreases with increased arrival rate).

our simulation can be used to predict the maximal number of clients that **QuickSOL** can efficiently handle when these client requests arrive at some expected arrival rates. As can be easily seen from the figures, even though we have a much higher arrival rate of client requests and **a fully loaded system**, **FindSOL** often responds faster than existing algorithms[19].

6 Related Work

Over the last few years, there has been growing interest in the efficient retrieval of multimedia data from disk.

Venkat Rangan et. al.[23] provides an excellent survey of the state of the art techniques used in video storage and retrieval. In this section we give an overview of the current research on the mutlimedia information retrieval from disk, and we show how the algorithms we provide differs from these approaches.

Work on Placement Mappings: Vin et. al.[24] develop two different ways of placing media data on disk arrays and analyze the performance of media systems based on these placement strategies. Similarly, Berson and Ghandeharizadeh[2] argue that staggered striping (a particular kind of placement mapping) enhances performance in video on demand systems. Ozden et. al. [17] propose movie on demand systems where each movie is stored on a separate disk which, of course, is a special kind of placement mapping. Our work complements these efforts – our **QuickSOL** algorithm can work with *any* placement policy whatsoever and hence may be used in conjunction with the approaches. Furthermore, our framework provides a general approach that can be used with arbitrary objective functions, rather than the specific objective functions. As a consequence, our model allows media-service providers to "plug in" different kinds of objective functions in an attempt to evaluate how well these objective functions match observed performance.

VCR Capabilities: Several authors [5, 7, 8] develop disk models that facilitate supporting different kinds of VCR operations such as fast forward, rewind, pause, etc. [5] provides ways of scheduling "rewind" operations. [7] develop techniques to support browsing video sequences at variable rates (thus implicitly supporting fast forward, rewind, etc.). They are interested in sampling segments of video and minimizing the variation on the number of vide segments skipped during fast forward/rewind. [8] develop a statistical model of vide retrieval that maximizes *statistical* quality of service under some conditions.

The observant reader would have noticed that all the above works optimize *different* objective functions, even though they all have the same common goal, viz. to efficiently support the standard VCR operations. Our paper provides a *single unified framework* within which different objective functions can be tried out. Furthermore, though [7] allows access from disk arrays (the other papers in this group do not), these arrays are assumed to be homogeneous. In contrast, in our paper, we have tried to provide a *single unified framework* within which different objective functions can be tried out under different placement mappings using heterogeneous disk servers.

Buffering and Prefetching Issues: Ng and his colleagues[12, 15] have developed efficient buffer management techniques to handle video data and ensure that the clients' consumption rate requirements are met. They show that pre-fetching techniques enhance performance and that multiple media-streams may neatly share buffers. Freedman and Dewitt[10] develop simulation studies showing that certain buffer page replacement and prefetching algorithms perform better than algorithms such as the group sweeping scheme algorithm of Yu et. al. These works are orthogonal to our main concerns in that these authors do not address the problems we address (or vice-versa) but we can certainly benefit from such algorithms. In effect, the use of methods such as prefetching decrease the delay time for clients and effectively behaves as though the bandwidth of the disk has increased. By "plugging" in this enhanced bandwidth rate in our model, our **QuickSOL** algorithm will be able to effectively satisfy more clients.

Other Efforts: Other authors such as [9] are developing comprehensive video servers including media players and file servers. Muntz and his colleagues [3] are developing performance metrics to assess fault tolerant construction of multimedia servers. Content-based methods to index and retrieve multimedia data have been developed in [1, 13] – these approaches develop models and representations of the information contained in multimedia data, but do not attempt to specify how to retrieve and presented uninterrupted streams of such data from disk which is the topic of this paper.

7 Conclusions

Over the last few years, there have been significant advances in multimedia technology. In particular, methods have been developed to efficiently access data from disk servers. However, even in the case of single disk systems and/or homogeneous disk arrays, different researchers have proposed different models, each with its own assumptions about placement strategies and objective functions on what to optimize.

In contrast, in this paper, we have provided an algorithm that can take as input:

1. a placement mapping that describes how movies are laid out on disks,
2. a specification of the capabilities of the various disk servers involved,
3. a specification of precisely what evaluation metric is used to measure the "good"ness of a particular way of accommodating user requests,
4. a specification of what events occur at a given time (including VCR like operations such as fast forward, rewind, etc.).

Our algorithm returns as output, an *optimal* new state that satisfies the user requests (if one exists). If none exists, the algorithm satisfies as many requests as possible, in a prioritized fashion (continuing clients get higher priority than new clients entering the system). Our algorithm is provably sound, complete and optimal.

We have implemented the **QuickSOL** algorithm and evaluated a number of different evaluation metrics and shown that the evaluation metric that maximizes total span seems to achieve the best performance in conditions when high load occurs. Our experiments also allow the user to specify his requirements (e.g. response time R and expected number of clients E) and using the experimental graphs, he may determine how many servers his application needs and how much buffer space his application needs in order to satisfy these requirements.

In this paper, we have provided a single unified framework within which we may reason about theoretically, and evaluate experimentally, methods for the retrieval of continuous media data in movie on demand systems. Our work applies to systems consisting of *heterogeneous disk servers* and applies to arbitrary ways of placing the data on the disk and arbitrary ways of evaluating states. Such state-evaluations reflect different objectives that a media-on-demand service provider may wish to accomplish. We have developed the **QuickSOL** algorithm that efficiently computes new states that handle a set of client transactions. **QuickSOL** is independent of placement mappings, and independent of objective functions, and hence, it can work with many different placement mappings [2, 18, 21, 24]. We have implemented our **QuickSOL** algorithm and conducted experiments to determine how many clients can be effectively serviced by **QuickSOL**. Furthermore, **QuickSOL** applies to many situations to which previous special-purpose algorithms do not apply, thus making it much more widely usable.

Acknowledgements. We thank Donald Kossmann for detailed comments on an earlier version of this manuscript.

References

[1] S. Adali, K.S. Candan, S.-S. Chen, K. Erol, and V.S.Subrahmanian. *Advanced Video Information Systems*, Accepted for publication in: ACM Multimedia Journal.

[2] S. Berson and S. Ghandeharizadeh. (1994) *Staggered Striping in Multimedia Information Systems*, Proc. 1994 ACM SIGMOD Conf. on Management of Data, Minneapolis, MN, pps 79–90.

[3] S. Berson, L. Golubchik and R. Muntz. (1995) *Fault Tolerant Design of Multimedia Servers*, Proc. 1995 ACM SIGMOD Conf. on Management of Data, San Jose, CA, pps 364–375.

[4] A. Brink, S. Marcus and V.S. Subrahmanian. Heterogeneous Multimedia Reasoning. IEEE COMPUTER, 28, 9, pps 33–39, Sep. 1995.

[5] H.J. Chen, A. Krishnamurthy, T.D.C. Little and D. Venkatesh. (1995) *A Scalable Video-on-Demand Service for the Provision of VCR-like Functions*, Proc. 1995 IEEE Intl. Conf. on Multimedia Computing Systems, pps 65–72.

[6] K.S. Candan, E. Hwang and V.S. Subrahmanian. (1995) A Unified Model for Continuous Media Data on Heterogeneous Disk Servers, Univ. of Md. Tech. Report, Oct. 1995.

[7] M.-S.Chen, D.D. Kandlur, and P.S. Yu. (1994) *Support for Fully Interactive Playout in a Disk-Array-Based Video Server*, Proc. ACM Multimedia 1994, pps 391–398.

[8] J.K. Dey-Sircar, J.D. Salehi, J.F. Kurose and D. Towsley. (1994) *Providing VCR Capabilities in Large-scale Video Servers*, Proc. ACM Multimedia 1994, pps 25–32.

[9] C. Federighi and L. Rowe. (1994) *A Distributed Hierarchical Storage manager for a Video-on-Demand System*, Proc. of the 2nd SPIE Symp. on Sotrage and Retrieval of Video Databases, pps 185–197.

[10] C. Freedman and D.J. Dewitt. (1995) *The SPIFFI Scalable Video-on-Demand System*, Proc. 1995 ACM SIGMOD Conf. on Management of Data, San Jose, CA, pps 352–363.

[11] D.J. Gemmell. (1993) *Multimedia Network File Servers: Multi-channel Delay Sensitive Data Retrieval*, Proc. ACM Multimedia 1993, pps 243–250.

[12] D. Makaroff and R. T. Ng. (1994) *Schemes for Implementing Buffer Sharing in Continuous Media Systems*, Information Systems, 19, 4, pps 33–54.

[13] S. Marcus and V.S. Subrahmanian. (1994) *Foundations of Multimedia Database Systems*, accepted for publication in: Journal of the ACM.

[14] G. Miller, G. Baber, and M. Gillilana. (1993) *News-on-Demand for Multimedia Networks*, Proc. ACM Multimedia 1993, pps 383–392.

[15] R.T. Ng and J. Yang. (1995) *An Analysis of Buffer Sharing and Prefetching Techniques for Multimedia Systems*, accepted for publication in: ACM Multimedia Journal.

[16] N.J. Nilsson. (1980) *Principles of Artificial Intelligence*, Morgan Kaufmann.

[17] B. Ozden, R. Rastogi and A. Silberschatz. (1994) *A Low-Cost Storage Server for Movie-on-Demand Databases*, Proc. 20th Conf. on Very Large Databases, pps 594–605, Santiago, Chile.

[18] D. Patterson, G. Gibson, and R. Katz. (1988) *A Case for Redundant Arrays of Inexpensive Disks*, Proc. ACM SIGMOD Conf. on Management of Data 1988.

[19] D. Rotem and J.L. Zhao. (1995) *Buffer Management for Video Database Systems*, Proc. 1995 IEEE Intl.Conf. on Data Engineering, pps 439-448.

[20] L. Rowe and B. Smith. (1992) *A Continuous Media Player*, Proc. 3rd Intl. Workshop on Networks and Operating Systems Support for Digital Audio and Video.

[21] K. Salem and H. Garcia-Molina. (1986) *Disk Striping*, Proc. 1986 IEEE Conf. on Data Engineering.

[22] P. Venkat Rangan, H. Vin and S. Ramanathan. (1992) *Designing and On-Demand Multimedia Service*, IEEE Communications Magazine, pps 56–64, July 1992.

[23] M. M. Buddhikot, S. S. Kumar, G. M. Parulkar, P. Venkat Rangan (1995) *Storage Hierarchies and Video Servers*, to appear in *Handbook of Multimedia Systems*, Prentice Hall, ed. Bill Grosky.

[24] H. Vin, S.S. Rao and P. Goyal. (1995) *Optimizing the Placement of Multimedia Objects on Disk Arrays*, Proc. 1995 IEEE Intl. Conf. on Multimedia Computing Systems, pps 158–165.