# Multimedia Storage Servers





# Multimedia Storage Servers

- Digitally stores heterogeneous data objects (consisting of audio, video, imagery, textual, and numeric data) on extremely high capacity storage devices
- Fundamental differences in data type characteristics and requirements
  - Best-effort service for text vs. real-time for video
  - Small read/writes for text vs. large read/writes for video
  - ....

# Approach

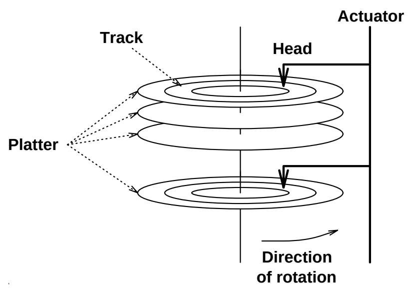
- Techniques for efficiently managing video data
  - Placement techniques
  - Fault tolerance issues
  - Scheduling, retrieval, and admission control
  - I/O stream sharing (buffering, batching, caching, ...)
- Techniques for designing integrated multimedia file systems
  - Architectural issues
  - Design and implementation of Symphony
- Methodology:
  - What are the fundamental issues ?
  - How to address these issues ? (Theory)
  - How to instantiate the solutions ? (Practice)





# Terminology

- Disk fundamentals:
  - Seek time
  - Rotational latency
  - Transfer rate
  - Scheduling algorithms: FCFS, SCAN, SSTF, SATF



## Terminology (Cont'd)

- Disk arrays
- Striping
  - Interleave the storage of each media stream among disks
  - Stripe unit: maximum amount of logically contiguous data that is stored on a single disk
  - Degree of striping: Number of disks across which a media stream is striped
- Redundant and non-redundant disk arrays





## Video Storage Server: Fundamentals

- Data transfer rate of disks $\gg$ data rate requirement of isolated video streams $\Rightarrow$ designing single-user video servers is straightforward
- Server stores digitized video streams on an array of disks
- Clients can request the retrieval of video streams for real-time playback
- Two possible server architectures:
  - Client-pull
  - Server-push

## Client-pull Architecture

- Server retrieves data only in response to an explicit request from client
- Used in conventional file system to provide *best-effort* service
- Adapting client-pull architecture for video: clients ensure playback continuity by
  - Determining the playback instant of a frame
  - Estimating response time for each request
  - Issuing a read request accordingly
- Response time: a function of the system load $\Rightarrow$ varies widely over time $\Rightarrow$ estimation is non-trivial





## Server-push Architecture

- Periodicity of video playback $\Rightarrow$ service clients in periodic *rounds*
- Round: retrieve a fixed number of frames for each media stream
- Continuous retrieval $\Rightarrow$ total service time must not exceed the playback duration of frames retrieved during a round

# Video Server Design Issues

- **Objective:** Maximize the number of clients that can be simultaneously supported
- Placement Issues
  - Determine the stripe unit size
  - Determine the degree of striping
  - Efficient failure recovery
- Retrieval Issues
  - Disk scheduling
  - Resource Reservation and admission control
  - Buffering and caching techniques





# Placement Techniques

## Efficient Placement on Disk Arrays

- Stripe video streams on disk arrays in terms of blocks (or stripe units)
- *Fixed-size Blocks*
  - Each block contains fixed number of storage units (i.e., bytes)
  - VBR compression $\Rightarrow$ variable number of frames per block
- *Variable-size Blocks*
  - Each media block contains fixed number of frames
  - VBR compression $\Rightarrow$ variable block sizes
- Key question: What should be the block size?





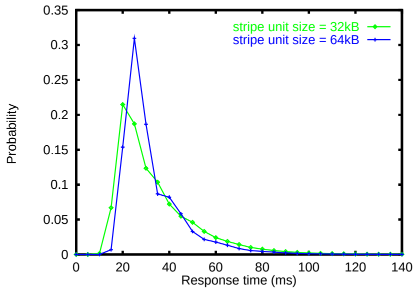
## Factors Affecting Optimal Block Size

- Server configuration
  - Number of disks in the array
  - Physical characteristics of disks
  - Type of the disk array (i.e., redundant vs. non-redundant)
- Client characteristics
  - Number of clients accessing the server
  - Client request size
- Objective: Given the server configuration and client characteristics, determine the optimal block size

# Selecting a Metric

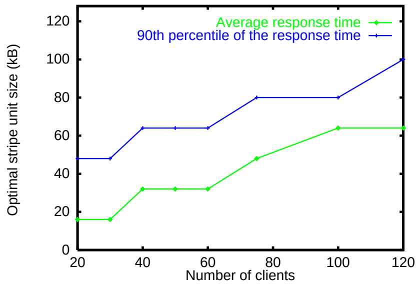
## Text

- Aperiodic accesses ⇒ client-pull architecture
- Best-effort ⇒ minimize response time




## Continuous Media

- Periodic and sequential accesses ⇒ server-push architecture
- Real-time ⇒ minimize tail of the response time distribution




- **Metric:** Minimize the service time of the most heavily loaded disk





# Fixed-Size Blocks

| | Small Stripe Unit Size | Large Stripe Unit Size |
|---|---|---|
| Load Imbalance | Small | Large |
| Latency Overhead | Large | Small |

- Choose a stripe unit size that balances these tradeoffs
- **Approach:** Develop an analytical model to compute the service time of the most heavily loaded disk
  - Non-redundant arrays
  - Redundant arrays

# Analytical Model

- **Given:** Server configuration and client characteristics
- **Objective:** Predict the service time of the most heavily loaded disk
- **Main Steps:**
  1. Estimate the number of blocks accessed from a disk by a single client
  2. Estimate the total number of blocks accessed from a disk
  3. Estimate the number of blocks accessed from the most heavily loaded disk
  4. Compute the service time of the most heavily loaded disk





# Analytical Model (Cont'd)

- Use frame size distribution to determine the total number of blocks accessed ($C_i$) from the array by a client (let $P(C_i = k) = b_i^k$)
- Number of blocks accessed from disk $j$ by client $i$ ($X_i^j$)
  - Property: A request is equally likely to start on any of the $D$ disks

$$\Rightarrow P(X_i^j = 1) = \frac{b_i^1}{D} + \frac{2 \cdot b_i^2}{D} + \cdots + \frac{D \cdot b_i^D}{D}$$

$$P(X_i^j = k) = \sum_{m=1}^{D} b_i^{(k-1)D+m} \cdot \frac{m}{D} \quad (k = 1, 2, 3...)$$

- Total number of blocks accessed from disk $j$ by $n$ clients

$$Y^j = \sum_{i=1}^{n} X_i^j$$

# Analytical Model (Cont'd)

- Number of blocks accessed from the most heavily loaded disk

$$Y^{\max} = \max(Y^1, Y^2, ..., Y^n)$$

- Service time of the most heavily loaded disk

$$\tau = E(Y^{\max}) * (t_s + t_r + t_x)$$

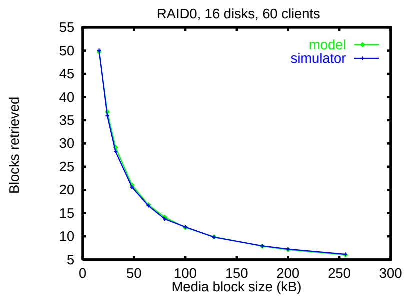
where

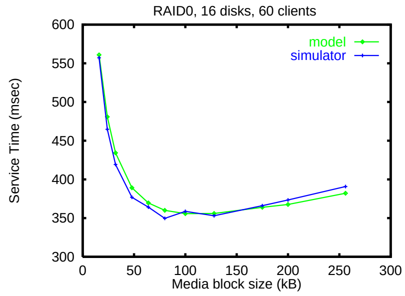
| | | |
|---|---|---|
| $t_s$ | : | seek time to access a block |
| $t_r$ | : | rotational latency to access a block |
| $t_x$ | : | transfer time of a block |

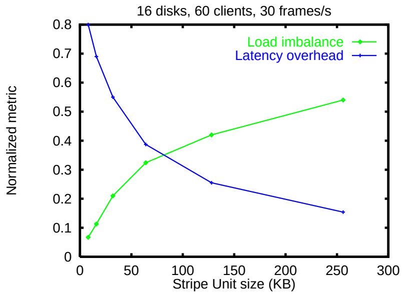




# Validation of the Model

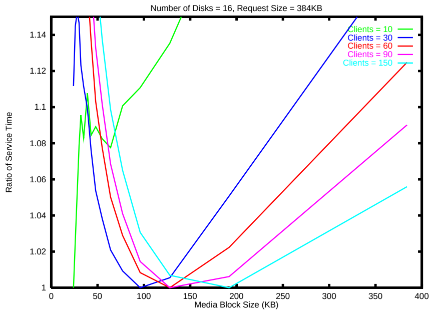


# The Optimality Envelope

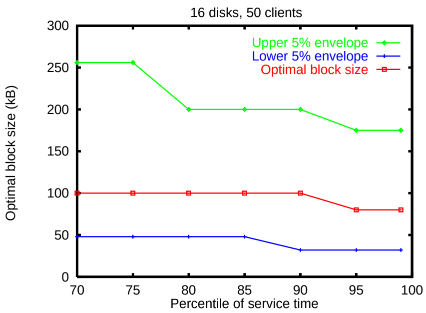
- Minima of the service time graph $\Rightarrow$ optimal media block size
- $x\%$ optimal envelope : Block sizes which yield service times within $x\%$ of the optimal





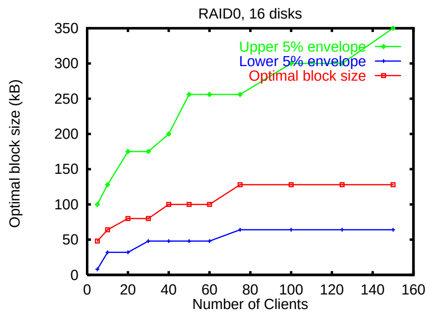


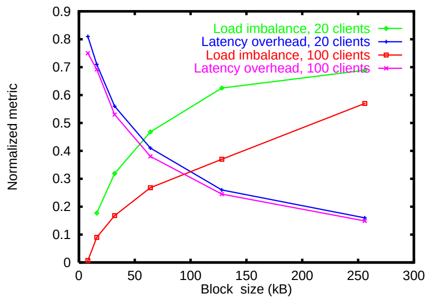

# Effect of Quality of Service Requirements




- Optimal stripe unit size decreases with increase in QoS requirements
- Ad-hoc techniques that choose a large stripe unit sizes can reduce the number of clients supported for stringent QoS!

# Effect of the Number of Clients



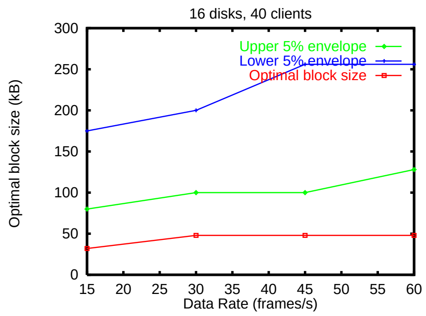




- Optimal stripe unit size increases with increase in number of clients
- Small number of clients: Small block size yields uniform distribution of load
- Large number of clients: Large block size yields higher disk throughput



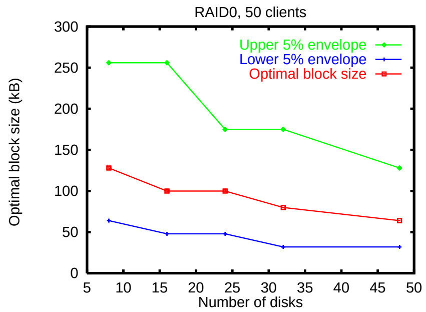


# Effect of Data Rate




- Optimal block size increases with increase in the data rate

# Effect of Number of Disks




- Optimal block size decreases with increase in the number of disks





# Selecting a Block Size

- Depends on design goal, server configuration, and client characteristics
- **Example:** Minimize service time over a range of workloads
  - Client characteristics such as data rate may be unknown at design time $\Rightarrow$ Block size must provide good performance over a range of such parameters
  - Determine the $x\%$ optimal envelope for different workloads
  - Choose a block size that is contained within the optimal envelope for *all* workloads

# Variable Block Sizes

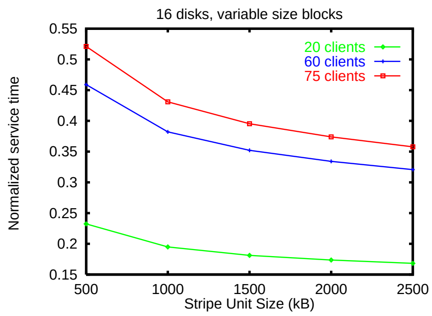
- Each block contains a fixed number of frames
- Minimizing seek and rotational latency $\Rightarrow$ store all frames accessed in a round in a single block
- Each client accesses exactly one block in each round
- Increase in the block size also increases the round duration and vice-versa





# Optimal Media Block Size

- Normalized service time (i.e., ratio of service time and round duration) decreases with increase in block size
- Optimal performance $\Rightarrow$ choose a block size as large as possible
- Tradeoffs:
  - Increased buffer requirement
  - Increased initiation latency



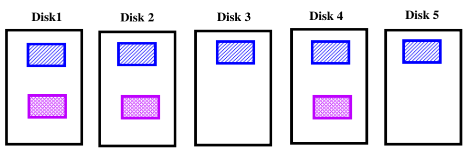
## Comparing Fixed-size and Variable-size Blocks

- Fixed-size Blocks:
  - Advantages:
    - Simple storage space management
  - Disadvantages:
    - Complex admission control
    - Load balancing is difficult
- Variable-size Blocks:
  - Advantages:
    - Retrieval of video streams proceeds in *lock-step*: load balancing possible at the time of admission
  - Disadvantages:
    - Inherent complexity in allocating and deallocating blocks
    - Higher implementation overhead



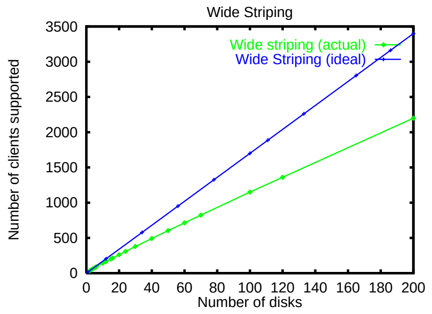


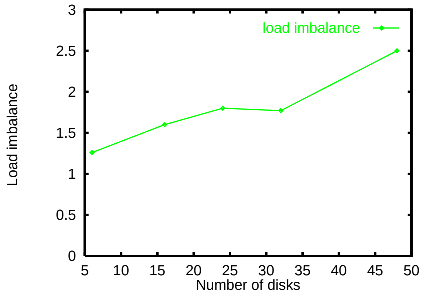
## Degree of Striping




- **Wide Striping:** Stripe across all disks in the array
  - Distributes the load across all the disks in the array
  - Prevents hot-spots
- **Narrow Striping:** Stripe across only a subset of disks in the array
  - Requires partitioning the array and mapping files to partitions so that load across partitions is balanced
- When is narrow striping useful ?

# Why Narrow Striping?




- Wide striping results in a loss in the number of clients supported for large disk arrays
  - Load imbalance increases $\Rightarrow$ utilization decreases
  - Optimal block size decreases $\Rightarrow$ throughput decreases





# Optimal Degree of Striping

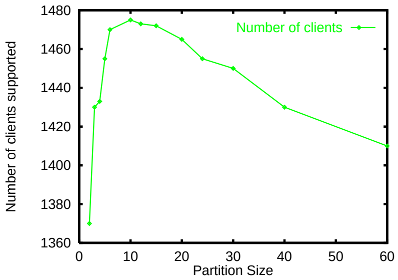
- **Approach:**
  - Partition the array and stripe each file across a sub-array
  - Allocate files to partitions such that each partition is equally likely to be accessed
- **Tradeoffs:**

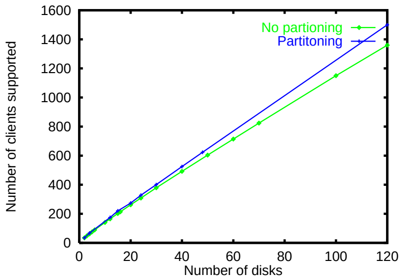
| | Imbalance within partitions | Imbalance across partitions |
|---|---|---|
| Small Partition Size | Small | Large |
| Large Partition Size | Large | Small |

- Choose a partition size that balances these tradeoffs

# Determining the Partition Size

- **Model:**
  - Compute the number of clients supported within a partition
  - Compute the number of clients supported by the entire array








# Mapping Files to Partitions

- **Objective:** Balance load across partitions
- Assume access frequency of objects is known
- Assign files to partitions such that total access frequency for all partitions is identical
- Replicate frequently accessed files on different partitions
- Dynamic load balancing and replication may be necessary

# Fault-Tolerance Techniques





# Fault Tolerance Requirement

- Increase in number of disks in the system $\Rightarrow$ increased frequency of disk failures
- Example:
  - Typical MTTF for conventional disks $\approx$ 200,000 hours $\Rightarrow$ Multimedia server consisting of 500 disks will see a failure every 400 hours
- Objectives of a failure recovery scheme:
  - Quickly rebuild information stored on failed disk, without taking the system off-line
  - Continue to meet the real-time performance requirements of video streams during failure recovery

# Failure Recovery in Disk Arrays: Fundamentals

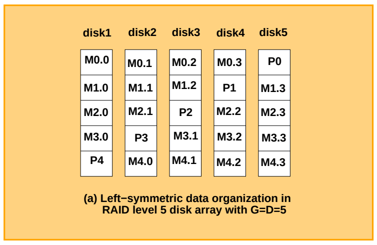
- **Mirroring:** 100% storage space overhead
- **Parity encoding:**
  - Recovery using redundant parity information
  - RAID-3:
    - Bit-interleaved parity
  - RAID-4:
    - Maintain the parity information on a dedicated disk
    - Limitation: parity disk becomes a bottleneck





# Failure Recovery in RAID-5

- **Basic idea:** Avoid hot-spot contention by scattering parity information across the array




(a) Left−symmetric data organization in RAID level 5 disk array with G=D=5

- **Disadvantages:**
  - If the system load is balanced prior to failure, then the surviving disks would observe 100% increase in load
  - Preventing saturation ⇒ low utilization in the fault-free state

# Declustered Parity Architectures

- Trade storage space for improved utilization/performance
- Approach:
  - Protect a smaller number of data blocks (say $(G-1)$) using each parity block
  - Distribute the parity information across all disks

| disk1 | disk2 | disk3 | disk4 | disk5 |
|---|---|---|---|---|
| M0.0 | M0.1 | M0.2 | P0 | M1.0 |
| M1.1 | M1.2 | P1 | M2.0 | M2.1 |
| M2.2 | P2 | M3.0 | M3.1 | M3.2 |
| P3 | M4.0 | M4.1 | M4.2 | P4 |
| M5.0 | M5.1 | M5.2 | P5 | M6.1 |

(b) Declustered parity organization with G = 4 and D = 5

- Failure recovery overhead $= \frac{(G-1)}{(C-1)}$



Department of Computer Science, UMass Amherst

# Observations

- Conventional failure recovery techniques:
  - Assume data as a sequence of bits, and do not exploit any data-specific characteristics
  - Assume perfect recovery requirement
- Characteristics of video:
  - Periodic and sequential access
  - Inherently redundant
  - Human perception tolerant to minor distortions in video playback $\Rightarrow$ imperfect recovery may be adequate

# Exploiting Sequentiality of Access

- **Approach:**
  - Compute parity across successive blocks of the same video stream
  - Utilize blocks accessed for failure recovery for playback and vice versa
- **Two schemes:**
  - *Lazy*: On accessing a block from a failed disk, retrieve the corresponding parity group
  - *Eager*: Prefetch blocks constituting a parity group and recover lost information prior to access
- **Tradeoffs:**
  - Advantage: Lower failure recovery overhead
  - Disadvantages: Higher buffer space requirement
  - Requires existing array controllers to be suitably extended





# Exploiting Inherent Redundancy in Video

- **Simple approach:**
  - Partition each image into sub-images such that none of the immediate neighbors of a pixel belong to the same sub-image
  - Store each sub-image on a different disk $\Rightarrow$ sub-image stored on the failed disk can be recovered by exploiting spatial redundancy
- **Observations:**
  - Conceptually simple and elegant
  - No additional failure recovery overhead on the disk array
  - Disadvantage: significant deterioration in compression efficiency
- Fundamental tension between:
  - Image partitioning, which tries to retain some redundancy for failure recovery; and
  - Image compression, which tries to eliminate redundancy for better compression

## Image Partitioning

- **Challenge:** Develop an image partitioning scheme that achieves effective failure recovery without any significant degradation in compression efficiency
- **Approach:** Partition an image *after* the inherent redundancies have been exploited by the compression algorithms
- Image partitioning process is dependent on the compression algorithm (or more specifically, on the transform functions)
- Example: Class of compression algorithms using Discrete Cosine Transform (DCT)



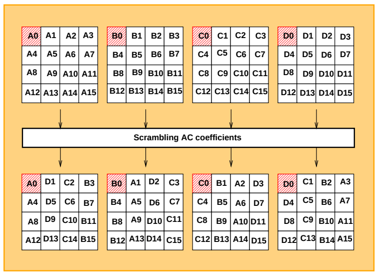


## Discrete Cosine Transform (DCT)

- DCT decorrelates an $n \times n$ pixel block, and redistributes the spectral energy amongst a small number of *transform coefficients*
- Application of DCT to a $n \times n$ block yields:
  - 1 DC coefficient, which captures the average color of the block; and
  - $(n^2 - 1)$ AC coefficients, which capture the details within the block
- Properties of DCT:
  - DC coefficients of the neighboring $n \times n$ blocks are correlated $\Rightarrow$ a DC coefficient can be approximated from neighboring DC coefficients
  - AC coefficients are uncorrelated and small in value $\Rightarrow$ quantization yields many zero AC coefficients

# Image Partitioning in the Transformed Domain

- Two steps:
  - **Scrambling**
    - ∗ Scramble the AC coefficients of $N$ DCT blocks to create $N$ scrambled DCT blocks



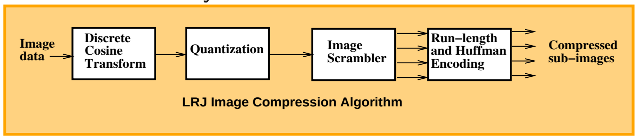

  - **Sub-image creation**
    - ∗ Partition the image into $N$ sub-images such that none of the DC coefficients in the 8-neighborhood of a block belong to the same sub-image





# Loss Resilient JPEG (LRJ) Algorithm

- Proceed row by row; select $N$ blocks at a time, scramble them, and assign to different sub-images
- Stagger the assignment of blocks of successive rows by at least 2 sub-images
- Implementation issues:
  - Padding with zero blocks, replication of Huffman tables
  - Effect on compression efficiency



## Reconstructed Images for $N = 2, 4, 8, 16$ in LRJ



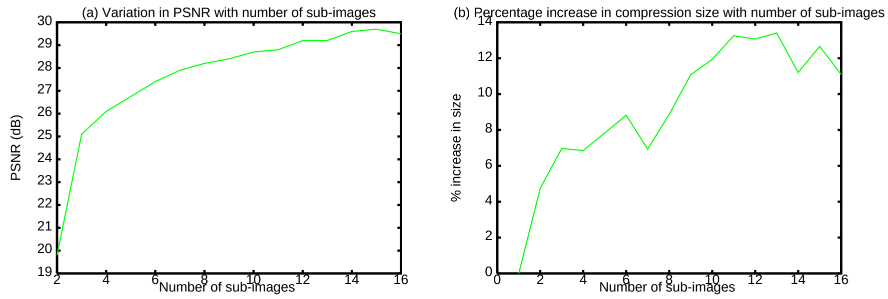


## Loss Resilient MPEG (LRM) Algorithm

- MPEG exploits both spatial and temporal redundancies
- Motion compensation produces error blocks and motion vector(s)
- Error blocks are transformed using DCT $\Rightarrow$ encoding and recovery technique for DCT blocks is identical to LRJ
- Motion vectors need to be perfectly recovered $\Rightarrow$ replicate motion vectors in sub-images

## Experimental Evaluation of LRJ

- Variation in the quality of reconstructed image (PSNR) and degradation in compression efficiency with increase in $N$








## Inherently Redundant Array of Disks (IRAD)

- Combine image partitioning technique with placement schemes
  - Disks storing the sub-images do not overlap
  - Failure recovery through extrapolation (LRJ / LRM )
- Perfect rebuild from backup storage or using parity-based recovery schemes
- **Advantages:**
  - No additional information is accessed during failure recovery
  - Integration of failure recovery with decompression $\Rightarrow$ inherently distributed and scalable architecture
  - Tolerates multiple disk failures and supports graceful degradation
  - Masks packet losses in network

## Discussion

- Exploiting the properties of data types yields efficient failure recovery schemes
- **Lazy and eager failure recovery schemes:**
  - Failure recovery overhead for declustered parity array: $1/(G-1)$
  - Higher buffer space requirement
- **IRAD:**
  - Inherently distributed, scalable, end-to-end, imperfect failure recovery architecture
  - Requires data type specific algorithms for recovery





## Retrieval Techniques

## Retrieval Techniques

- **Objective:** maximize the number of clients that can be serviced simultaneously
- **Approach:**
  - Employ *disk scheduling algorithms* to efficiently access blocks
  - *Reserve server resources* to ensure that the real-time playback requirements of clients are met





## Disk Scheduling Algorithms

- FCFS, SCAN, SATF,...
- Real-time scheduling algorithms: EDF
- Combined algorithms:
  - SCAN-EDF: Schedule requests with same deadlines in SCAN order
  - .....

## Resource Reservation

- Continuous playback requirement: Service time for $(k_1 + k_2 + \cdots + k_n)$ blocks $\leq \mathcal{R}$ where $\mathcal{R}$ denotes the duration of a round
- Basic approach:
  - Estimate the resource requirement of the new client;
  - Estimate the total resource requirement assuming that the client is admitted; and
  - Admit a new client only if the total resource requirements can be met
- Precise formulation of the admission control criteria depends on the service requirements of the clients





## Service Requirements

- Clients of a multimedia server can be either *tolerant* or *intolerant*
- Intolerant clients require deterministic service guarantees
- Tolerant clients can tolerate brief distortions in playback continuity
- Classification of admission control algorithms:
  - Deterministic: provides strict guarantees
  - Statistical: provides statistical guarantees
  - Predictive: provides acceptable performance but no absolute guarantees

# Deterministic Admission Control Algorithm

- **Approach:**
  - Utilize worst-case estimates for the number of blocks to be accessed and their relative positions on the disk
  - Ensure that the continuous retrieval requirements of clients are strictly met
- **Admission control criteria:**

$$k * (t_s + t_r + t_x) \leq \mathcal{R}$$

| | | |
|---|---|---|
| $k$ | : | maximum number of blocks accessed in a round |
| $t_s$ | : | worst case seek time to access a block |
| $t_r$ | : | worst case rotational latency to access a block |
| $t_x$ | : | transfer time of a block |

- **Observation:** Worst-case estimation of resource requirements $\Rightarrow$ low utilization





# Statistical Admission Control Algorithm

- Service time $\tau$ may vary from one round to another due to:
  - Variation in the number of blocks to be accessed during a round, and
  - Variation in their relative placement on disk
- **Key concepts:**
  - Utilize distributions (rather than the worst-case values) for service times and number of blocks to be accessed from disk; and
  - Provide statistical service guarantees to each client
- **Admission control:**

$$q * \mathcal{F}_o + (1 - q) * \sum_{i=1}^{n} f_i \geq \sum_{i=1}^{n} p_i * f_i$$

| | | |
|---|---|---|
| $\mathcal{F}_o$ | : | number of frames guaranteed to be accessed in a round |
| $q$ | : | overflow probability where $q = P(\tau > \mathcal{R})$ |
| $p_i$ | : | loss affordability of client $i$ |

## Statistical Algorithm (Cont'd)

- Computing the overflow probability $q$:

$$q = P(\tau > \mathcal{R}) = \sum_{k=k_{min}}^{k_{max}} P(\tau_k > \mathcal{R}) P(\mathcal{B} = k)$$

$\tau_k$ : time to retrieve $k$ blocks
$B$ : number of blocks retrieved in a round

- Determination of $\mathcal{F}_o$:
  - Determine the minimum number of blocks (say $k_d$) that are guaranteed to be accessed during each round of duration $\mathcal{R}$
  - Determine the minimum number of frames that can be contained in $k_d$ blocks



Department of Computer Science, UMass Amherst

## Predictive Admission Control

- Admit a new client if the measurements of the server load from the recent past indicate that the service requirements of clients can be met
- Admission control criteria:

$$(\widehat{k}_{new} + \sum \widehat{k}_i) * \widehat{\eta} \leq \mathcal{R}$$

$$\widehat{\eta} = E(\eta) + \epsilon_1 \cdot \sigma(\eta)$$

$$\widehat{k}_i = (E(k_i) + \epsilon_2 \cdot \sigma(k_i)) * p_i$$

$k_i$ : number of blocks accessed by client $i$
$\eta$ : time to retrieve a media block
$\sigma(x)$ : standard deviation of x

- Determine :
  - $E(\eta)$ and $\sigma(\eta)$ through measurement
  - $E(k_i)$ and $\sigma(k_i)$ from precomputed statistics

## Policies for Enforcing Requirements

- Aggressive admission control algorithm $\Rightarrow$ occasional violations of continuity requirements (i.e., $\tau > \mathcal{R}$)
- Key observation: sequential nature of video playback enables the server to predict overload
- **Overload control:**
  - Select a set of blocks to be discarded/delayed based on the loss affordability of clients as well as the relative placement on disk
  - Objective:
    - Minimize the number of blocks discarded/delayed
    - Distribute such blocks among tolerant clients



Department of Computer Science, UMass Amherst

## Overload Control Algorithm

- Compute loss affordability of client $i$ during round $j$ as $\chi_{i,j} = L_{i,j} - l_{i,j}$ where:

$$L_{i,j} = \lfloor j * (1 - p_i) * f_i \rfloor$$

  and

$$l_{i,j} = \sum_{m=1}^{j-1} \widehat{l_i^m}$$

  where $\widehat{l_i^m}$ denotes the number of frames of client $i$ discarded during the $m$th round

- Using $\chi_{i,j}$, label each media block to be retrieved during the round as either *can-be-discarded* or *can-not-be-discarded*
- Discard blocks in an order that maximizes the reduction in service time per discarded frame

# Techniques for Video Caching





# Techniques for Increasing Server Capacity

- Batching
- Caching
  - Interval caching
  - Frequency caching
- Key point:
  - In conventional systems, caching is used to improve program performance
  - In video servers, caching and batching are used to increase the server capacity

## Batching Policies

- Group clients requesting the same video object that arrive within a short duration of time or through adaptive piggy-backing
- Increasing batching window increases the number of clients being serviced simultaneously, but also increases reneging probability
- Performance metrics: latency, reneging probability and fairness
- Policies:
  - FCFS
  - MQL (maximum queue length)
  - FCFS-n: schedule the playback of n most popular videos at predetermined regular intervals, and service the remaining in FCFS order





## Interval Caching

- Exploits sequential nature of multimedia accesses
- Cache only the interval between two temporally spaced clients
- Algorithm:
  - Order the intervals based on increasing space – smaller interval implies smaller time to reaccess
  - Optimal for homogeneous clients
- Dynamically adapts to changes in workload

## Frequency Caching

- Typical video accesses follow the 80-20 rule (i.e., 80% of the requests access 20% of the video objects)
- Cache the most frequently accessed video objects
- Requires large buffer space
- Not dynamic – frequency determination is based on past history or future estimates





## Taxonomy of Cache Replacement Policies

- Recency of access: LRU, LRU-k
  - Locality of reference
- Frequency based: LFU, FBR
  - Hot sets with independent accesses
- Optimal: Time-to-reaccess (TTR)
  - Knowledge of the time of next access
- Size-based: LRU-MIN, GD-size
  - Different size objects
- Miss cost based: GD-size
  - Different time to fetch objects
- Resource-based caching
  - Resource usage of different object classes

# Resource-Based Caching (RBC)

- Select granularity of entity to cache
- Determine entity to replace based on:
  - Performance metric (e.g., hit ratio)
  - State of the cache ($U_s$, $U_b$)
  - Resource usage of the object $< s_i, b_i >$
- Based on two constraint 0/1 knapsack
  - Greedy approximation
  - Mappings to multiple single constraints
- Example:
  - Performance metric byte hit-ratio
  - Space constrained cache
    - ∗ Order by time-to-reaccess