MEDIC: A Memory & Disk Cache for Multimedia Clients

Edward Chang and Hector Garcia-Molina
Department of Computer Science
Stanford University
{echang, hector}@cs.stanford.edu

Abstract

In this paper we propose an integrated memory and disk cache for managing the resources of a multimedia client's computer or set-top box. The cache cushions the multimedia decoder from input rate fluctuations and mismatches, and because data can be cached to disk, the acceptable fluctuations can be very large. This lets the client operate efficiently at low cost when the network rate is much larger or smaller than the media display rate, and gives the media server much greater flexibility for balancing workload and supporting VCR-like functions. We analyze the memory requirements for this cache, and analytically derive safe values for its control parameters. Using a realistic case study, we examine the interaction between memory size, peak input rate, and disk performance, and show that a relatively modest amount of main memory can support a wide range of scenarios.
Keywords: Multimedia client, resource management, disk scheduling, memory management.

1 Introduction

The data for a multimedia presentation (i.e., video and audio) is delivered via a channel from a server to a client. To date, most research has focused on the design of the media server [9, 10, 11, 12, 13, 17, 22, 24, 27, 28, 30, 31], while the media client has received little attention [7, 19, 23, 26]. Most research assumes that the client simply has to play back the data as it receives it.

In this paper we do focus on the client side, presenting a combined memory-disk buffering algorithm that allows the client to dynamically and effectively deal with variable data rates and delays. We call this algorithm *MEDIC*, for MEmory-DIsk Cache. MEDIC manages the resources of the client's computer or set-top box. MEDIC carefully allocates its limited memory to competing tasks, i.e., to receiving new data from the network, to writing data to disk as memory fills up, to reading data from disk as needed, and to holding data for decoding and playback. Since data is concurrently written to the disk cache and read from the disk cache, MEDIC must also intelligently issue IOs to avoid undue conflicts.

To examine the need for MEDIC, we need to address three questions: Why is a cache needed at the client side? Why should the cache include disk space? Why do we need an integrated memory-disk cache?

The answer to the first question is fairly obvious: a client cache is needed as a cushion against variability in the data delivery rate, and against differences in delivery and consumption rates. If one assumes a perfect media server, which can pump out the data at precisely the rate it will be consumed by the client, and one assumes a perfect network, which can deliver the data at this same rate, then one does not need a buffer at the client. However, the server and network may deliver data at a rate different from that at which data is consumed, and the rate may not be constant, due to network disturbances, traffic congestion, router failures, server glitches, and so on. Thus, a client cache is *necessary* to handle the delivery-consumption mismatches. For example, if the delivery rate is smaller than the consumption rate, the cache must save up enough data before playback starts. Similarly, the cache can provide playback data to the decoder during input lulls. Furthermore, a client cache is also quite *useful*, in that it makes the requirements on the server and network less stringent: the larger the cache, the more flexibility is given to the server and network for data delivery.

A cache can be made larger by adding disk storage, and hence the usefulness of a *disk* cache. A disk cache at the client gives the server much more delivery flexibility. For instance, a server can then send much more data to the client during a low utilization period (if the network permits), and less data during a peak utilization period. In this sense, the client assists in server load balancing. For example, say, a server with a capacity to service 100 clients expects its peak load in 30 minutes. If the server is currently servicing 50 clients that have 60 minutes more of playback time, 50% of server's capacity will be tied up during the peak period. If the channel capacity is sufficient, the server can double the data delivery rate to the current 50 clients, freeing capacity for the peak period. For this to work, the client must have a large cache, and using a disk for this large cache (as opposed to main memory only) significantly reduces costs.

A large disk cache also makes it possible to deal with limited channel capacity. For instance, say a news clip that has a display rate of 1.5 Mbps is delivered via a 1.0 Mbps channel to the client. Suppose the playback duration of the news clip is 1000 seconds (about 16 minutes). During playback, the client only receives 1000×1.0 Mb, but consumes 1000×1.5 Mb. Thus, when the clip is started, the client needs to have cached at least 1000×0.5 Mb, or 62 Mbytes. The cost of saving this data on disk is a lot lower than saving it all in main memory!

In addition, some VCR functions can be supported more can efficiently if the client can buffer a large amount of data. For instance, when a pause command is initiated by the viewer, the client can continue receiving bits into its local disk without disrupting the server. Another example is the VCR command *rewind*. If the client keeps a copy of the presentation in the local disk, the rewind operation can be performed locally without interfering with the server's schedule. Without client assistance, the pause and rewind operations prolong playback time and may significantly decrease server throughput.

Our third question deals with the need to integrate the memory and disk caches. In some current systems, a disk cache is used to download entire presentations, and then a memory cache is used for playback (from the local disk). In our earlier news clip example, we would need 25 minutes (1500 seconds) to download the entire clip, before starting the presentation.

However, if the disk and memory caches are well integrated, playback can start (from the local disk) as soon as the initial 62 Mbytes arrive, i.e., in about 8 minutes (500 seconds). While data is being played back from the disk, new data arrives, and is written to disk. Algorithm MEDIC performs this type of buffering in an adaptive fashion: If there is enough memory at a given point, data need not go to disk; it can go directly to the decoder. MEDIC also gracefully deals with out of order packets, using its memory or disk buffer to correctly sequence the data given to the decoder. (It is best to let MEDIC deal with packet order instead of assuming that the network layer provides a reliable stream, because MEDIC knows precisely the instantaneous playback needs and can thus be much more effective.)

In summary, the contributions of this paper are as follows:

- We present algorithm MEDIC, an adaptive, integrated memory-disk buffer manager. The challenge is to develop an algorithm that
  1. can efficiently deal with the competing and unpredictable read and write memory requirements,
  2. avoids writing data to disk when it is not necessary,
  3. handles out of order network data,
  4. works on any off-the-shelf client disk, and
  5. has the right *control parameters* for tuning.
- We derive bounds for the amount of memory a client needs to support given input and playback characteristics. From these bounds we compute effective settings for the control parameters, which determine the timings and sizes of disk IOs.
- We present a realistic case study, and show how input peak rates, disk performance, and other parameters affect memory and performance. Our results indicate that, thanks to the disk cache, a relatively modest amount of main memory can support a wide range of input and media rates.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 models the data input and consumption functions. Section 4 describes the data structures and parameters used by algorithm MEDIC. In Section 5 we prove various properties of algorithm MEDIC, leading to memory use bounds and good control parameter settings. Section 6 evaluates algorithm MEDIC's performance and suggests methods for improvement. Finally, we offer our conclusion in Section 7.

2 Related Work

Many studies have proposed rate-adaptation mechanisms that adjust the media delivery rate at the server by explicit quality-of-service (QoS) feedback from the client [7, 19, 23, 26]. To our knowledge, no scheme for adaptive buffer management at the media client side has yet been proposed. We deem an adaptive and "intelligent" buffer management policy at the client side orthogonal and complementary to the rate-adaptation mechanisms that the server employs, and argue that together the two can provide a complete end-to-end solution for media delivery.

Using memory-disk integrated caches at the client side has been explored for conventional file systems [18] and client-server database systems [16]. However, the problems that a multimedia system faces are different. First, the media data must meet display deadlines. Conventional buffer management schemes focus primarily on reducing response time and do not take real-time constraints into consideration. In addition, the access pattern for media data is very different: once a page is decoded, it is not used again (except if we support a rewind feature). Thus, page replacement policies such as LRU and FIFO, which aim to maximize the hit rate are not applicable to MEDIC.

While a number of studies have addressed real-time transaction and IO scheduling [4, 5, 6, 8, 21], to our knowledge no work has dealt with memory-disk cache management for multimedia clients. The work that is most relevant is reported in [25]. That work proposes a Priority Memory Management (PMM) algorithm that minimizes the number of missed deadlines by adapting both the multiprogramming level and the memory allocation strategy of a real-time database system according to feedback on system behavior. Like other real-time scheduling policies, PMM meets deadlines for high priority jobs with high probability at the expense of the low priority jobs. MEDIC, on the other hand, must guarantee a continuous supply of media data to the decoder. Another difference between traditional real-time policies and MEDIC is that the traditional ones allow tasks to be completed ahead of their deadlines, while for MEDIC early delivery of data to the decoder may unnecessarily fill up memory. Thus, MEDIC prefers a data supply strategy that delivers data not only in time, but *just in time* (JIT). Furthermore, MEDIC does not assume a real-time disk scheduler.

In addition to managing buffers to meet real-time constraints, MEDIC must also deal with channel delays. Protocols such as IP, UDP, and ST2 (ST2 is a real-time IP layer equivalent protocol) do not guarantee in-order packet delivery or reliable transmission [1]. Multimedia applications are usually implemented directly on top of these protocols because reliable protocols can interfere with media delivery [3]. For example, an error correction scheme such as the automatic retransmission query protocol (ARQ, and of which TCP is a typical implementation) tends to aggravate the media delivery problem for at least three reasons:

- ARQ may interrupt the media server's regular schedule to re-fetch already transmitted data from a secondary storage device. This creates an extra bandwidth requirement that most server architectures cannot handle well.
- ARQ does not work well with data transmission with real-time requirements since the retransmission requests may further slow down the networks and the retransmission of the missing data may well miss its deadline.
- ARQ can be catastrophic if multicast is involved since any network delay or errors can induce a flood of retransmission requests and retransmissions, which defeats the purpose of multicast.

For these reasons, we assume unreliable delivery of data in the research presented in this paper, and if necessary use the MEDIC buffers to place data in the correct sequence.

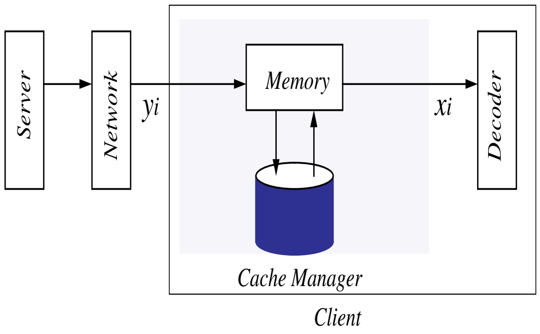


Figure 1: Media Data Delivery System Components

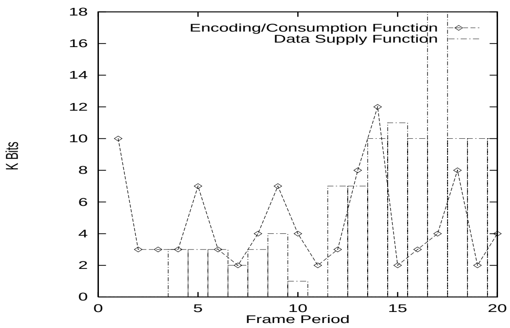


Figure 2: Functions

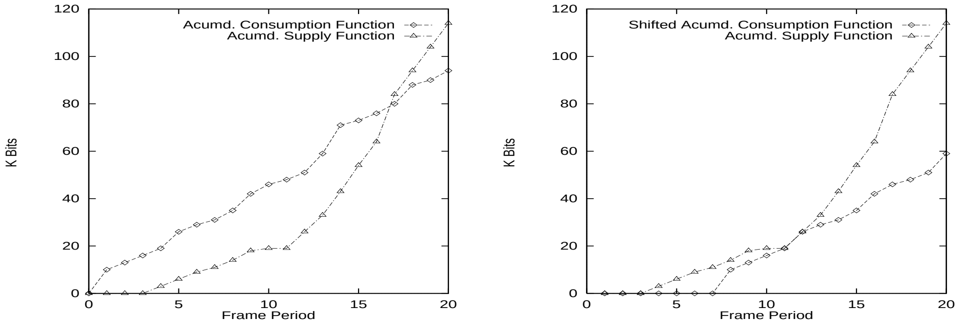
3 Model

In this section, we model three major parameters in a media data delivery system: the data input rate, the data consumption rate, and the initial latency.

Figure 1 depicts a media data delivery process. First, the media data is encoded and stored in the server's storage device (on the left-hand side of the figure). The encoded bit stream can either be at a variable bitrate (VBR) or at a constant bitrate (CBR). We define the encoding function (and thus decoding function in Figure 1) as x_i, where i denotes the i^{th} frame period, and x_i the number of bits used to encode the i^{th} frame. For instance, a movie that lasts 100 minutes, with 25 frames per second, consists of $100 \times 60 \times 25 = 150,000$ frame periods, each lasting 40 milliseconds. The points in Figure 2 show an encoding function for an MPEG2 movie. Since the number of bits required for encoding the anchor frames (i.e., the I and P frames) is larger than that for the intercoded frames (i.e., the B frames), we see that x_i fluctuates from period to period.

The decoder (on the right-hand side of Figure 1) uses an "inverse" scheme for playback, so x_i is also the data consumption function. (The decoder can infer how much data is needed at each point of time from the stream itself. For a detailed specifications of some coding standards, such as H.261 and H.263, please consult references such as [2].)

Data can be delivered to the client with a delivery profile y_i that may differ significantly



(a) Before Delay

(b) After Delay

Figure 3: Accumulated Rates

from x_i. (The amount of data arriving in frame period i is given by y_i.) For example, data can be packetized and delivered to the client using a constant size and rate packetization (CSRP) scheme, a variable size and rate packetization (VSRP) scheme [15], or a combined scheme (e.g., constant size, variable rate or variable size, constant rate). Furthermore, the data delivery rate can be faster or slower than the data consumption rate depending on the channel capacity. Finally, errors may also affect the delivery profile. The rectangles in Figure 2 show a sample y_i function.

The cache manager at the client must "convert" the y_i input into the x_i decoder consumption function. However, the x_i function needs to be shifted in time by some amount ψ (the initial playback delay), so that there is enough data available. To illustrate, Figure 3(a) shows the *cumulative* x_i and y_i functions, based on the original data in Figure 2. To ensure that the data received is always greater than or equal to the data consumed, playback must be delayed by at least 7 time frames. Figure 3(b) shows x_i shifted by 7 units, and, indeed, we see that now there is enough data at all times.

The following inequality describes this continuous data supply requirement quantitatively:

$$\sum_{i=1}^{k} y_i \geq \sum_{i=1}^{k} x_{i-\psi}, for\ \forall 1 \leq k \leq tt + \psi, \tag{1}$$

where tt denotes the total number of frame periods of x_i, and $x_i = 0$ and $y_i = 0$ for $i \leq 0$. The minimum storage required by the client is the maximum difference between the accumulated value of y_i and the accumulated value of x_i over all $i's$, or

$$\max\{\sum_{i=1}^{k} y_i - \sum_{i=1}^{k} x_{i-\psi} | for\ \forall 1 \leq k \leq tt + \psi\}.$$

We have argued that this amount can be quite large, so a disk cache can be very cost effective.

4 The MEDIC Algorithm

This section describes the MEDIC client cache algorithm. Its function is to initially accumulate data, and then supply the decoder with the x_i profile, using a limited amount of main memory.

A simple cache manager could be implemented with two separate buffers, one for receiving network data and writing it to disk, and another for reading disk data and giving it to the decoder. However, our goal is an integrated cache, where buffer pages can be used for what is most critical at the moment. In this environment where memory is not reserved in advance for particular tasks, it is important to avoid conflicts, e.g., not being able to allocate an empty buffer for data coming off the disk, because memory is full holding data waiting to be written to disk. We also want to avoid writing data to disk at all, when there is enough memory to hold it for playback.

4.1 Resources and Data Structures

The memory buffer is composed of memory pages, and the disk buffer of disk blocks. Both buffers are composed of a free pool and an in-use pool. When a memory page or a disk block is allocated, it is taken out of the free pool and marked in-use. When a memory page or a disk block is released, it is returned to the free pool. To simplify the discussion, we assume the page size is identical to the block size. Since for typical clients it is hard to change the disk block allocation strategy, in this paper we assume that blocks containing sequential data may *not* be contiguous on disk. (In Section 6 we show the gains achievable if blocks were contiguous.)

When a packet arrives at the client, MEDIC receives the packet immediately in its private work space to prevent the network buffer from overflowing. The resource manager first checks if the packet has passed the display deadline. If it has, it is discarded.[1] Otherwise, the resource manager allocates memory pages from the free memory pool to stage the packet. (If the page size is larger than the packet size, a page is allocated for multiple packets.)

MEDIC assembles the arrived data into a virtually contiguous bit stream. Figure 4 shows the MEDIC page table. It consists of a list of page table entries. Each entry consists of a valid field and an address field. If a page is valid (i.e., the page has arrived and has not yet been consumed by the decoder), the valid bit is set. For a valid page, the address field in its page table entry indicates where the page physically resides, that is, in the memory buffer or disk buffer. We denote the virtual pages of a bit stream p_1 to p_n, where n is the total number of pages in the stream. We denote the physical page corresponding to p_i as m_i if it resides in memory, and as d_i if it resides on disk. Figure 4 shows an example where pages p_1 and p_2 are memory resident, and pages p_3 and p_4 disk resident.

The example of Figure 5 shows how MEDIC handles packets that are larger than a page, and packets that are out of order. For the example, assume that the packet size is 1.5 memory pages. Figure 5(a) shows the situation when packet k_2 has arrived ahead of packet k_1. The

[1] Such missed deadlines should not occur in a multimedia system. However, since MEDIC has no control over the input stream, it cannot assume these undesirable events will not occur.

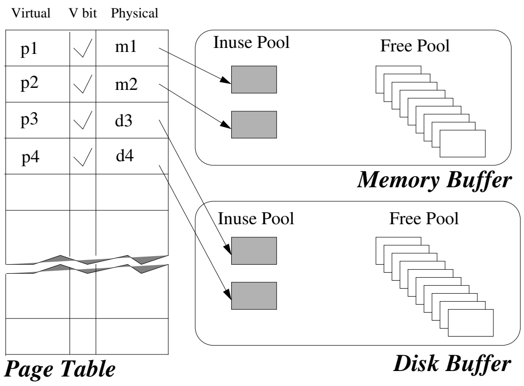
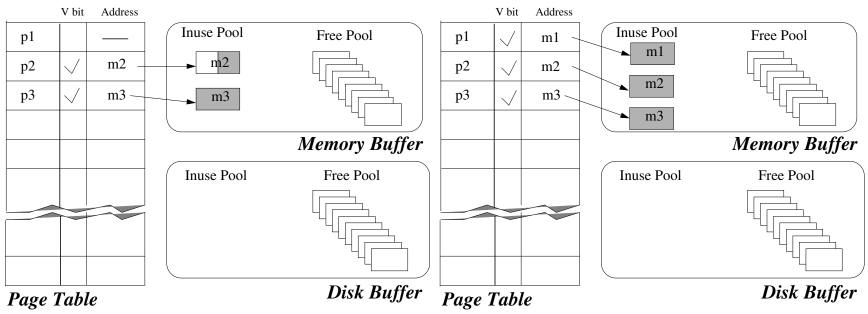


Figure 4: Page Table Structure



(a) When $k2$ arrives (b) When $k1$ arrives

Figure 5: Page Table Entries

k_2 data is copied into the second half of the first allocated page (denoted as p_2 since it is the second virtual page in the bit stream) and onto the entire second allocated page (denoted as p_3). Note that page p_1 is not valid and the first half of p_2 (physically m_2) is empty, for packet k_1 has not yet arrived.

When k_1 arrives, MEDIC allocates only one memory page m_1 for p_1, because the physical page for p_2 is already valid in the page table. MEDIC copies the data of k_1 to p_1 and to the first half of p_2, then updates p_1's page table entry, as indicated in Figure 5(b).

MEDIC must decide when playback can start, when pages should be moved to disk, and when pages should be read from disk. Its policies for this are dictated by the following control parameters:

- $\sigma_{playback}$, the playback threshold: The number of pages to accumulate at the client before playback starts. These pages should contain the data needed at the beginning of the presentation.
- σ_{read}, the read threshold: If fewer than σ_{read} pages are ready for the decoder in memory (containing the next data for playback), then a read IO is initiated.

Parameter	*Description*
$\sigma_{playback}$	Playback threshold, in pages
σ_{read}	Read threshold, in pages
σ_{write}	Write threshold, in pages
ρ_{read}	No. of pages to make ready
ρ_{write}	No. of pages to write
M_{avail}	Available memory space, in pages
TR	Disk transfer rate, in Mbps
k_{size}	Packet size
p_{size}	Page size
k_i	The i^{th} packet
p_i	The i^{th} page of data
$\gamma(d)$	Function computes worst disk latency
S_r	Worst disk latency for a read request
S_w	Worst disk latency for a write request
T_r	Time a read guaranteed to be completed
T_w	Time a write guaranteed to be completed
δ	Frame period
ψ	The client's initial latency, in frame periods
tt	Playback time of a video, in frame periods
x_i	The media data encoding function
y_i	The media data supply function
θ	The page the decoder is currently accessing
M_d	No. of pages available to the decoder
M_f	No. of free pages available to stage packets
β_{disk}	No. of pages on disk

Table 1: Parameters

- ρ_{read}, the pages to read: When a read IO is initiated, ρ_{read} pages are requested. As explained below, in some cases fewer than ρ_{read} may be read.
- σ_{write}, the write threshold: If fewer than σ_{write} memory pages are available for incoming data, then a write IO is initiated.
- ρ_{write}, the pages to write: When a write IO is initiated, ρ_{write} pages are written out. All write IOs are for this amount.

To assist the reader, Table 1 summarizes these parameters, together with other parameters that will be introduced later. The first tier of Table 1 lists the control parameters. The second tier describes the physical and derived characteristics of the hardware resources. The third tier shows the parameters and functions that depict the data supply and input rates. Finally, the fourth tier presents parameters that keep track of the buffer usage.

We now describe informally how MEDIC operates given its control parameters, while Appendix A shows the MEDIC algorithm in more detail. MEDIC is an event-driven algorithm, where there are three types of events: a packet arrives, the decoder consumes a page, and an

IO completes. At each event, MEDIC updates its state variables to reflect the current situation and takes appropriate action. The two main state variables are:

- M_d, the number of pages available to the *decoder*, and
- M_f, the number of *free* pages.

First let us consider how MEDIC manages M_d and M_f when there are no partially filled pages and data arrive at the client in order. If page p_θ is being consumed, then M_d is the number of memory-resident pages that logically follow p_θ. If p_j is the last of these pages, and p_{j+1} arrives from the network, then it is immediately counted in the M_d pages.

If p_{j+1} is on disk, and data for a page that follows it arrives, then it is placed in memory but M_d is not incremented. When p_{j+1} and the pages that follow it are read from disk, M_d is incremented to reflect the latest sequential page in memory. For example, say p_3, p_4, are on disk, p_2, p_5, p_6 are in memory, and p_1 is being decoded. If p_3 and p_4 are read from disk, then M_d is incremented by 4, since now we have the next 4 pages in sequence in memory.

Of course, as pages are consumed by the decoder, M_d is decremented. Similarly, as any page in memory is allocated (for whatever reason), M_f is decremented, and as pages become free, it is incremented.

A write IO is triggered if $M_f \leq \sigma_{write}$ and no previous write IO is outstanding. The ρ_{write} pages to flush to disk are selected by our LSF (Late in the Sequence First) page replacement policy. This policy selects the later pages in the logical sequence. For example, if p_3, p_4, p_7, p_8 are in memory and $\rho_{write} = 2$, then we write p_7, p_8 to disk, since they will be needed by the decoder later than p_3, p_4.

A read IO is triggered if $M_d \leq \sigma_{read}$ and no other read IO is in progress. Two important issues should be noted about reads. First, a read may be for fewer than ρ_{read} pages. This can happen if there are fewer than ρ_{read} pages on disk, or if fewer pages are needed to increment M_d by ρ_{read}. To illustrate what happens if fewer pages are needed, say p_1, p_4, p_5 are in memory, p_2, p_3 are on disk, $\rho_{read} = 4$, and p_1 is the last page available to the decoder. In this case we need to read only p_2 and p_3 to make the next 4 pages available to the decoder, so that is all we read.

The second important issue is a subtle one and regards memory allocation for read IOs. The following example illustrates what can happen if we allocate this memory naively. Assume that $M_{avail} = 8$ and all control parameters are set to 2. In particular, σ_{write} is set to 2 because we estimate that we may receive up to 2 new data pages while a write IO is in progress. That is, at the beginning of a write IO we want to have at least 2 free pages to hold new data that may arrive while we flush data. Initially, pages p_2, p_3, p_4, p_5, p_8 are in memory, and p_6, p_7 on disk. At this point $M_d = 4$ and $M_f = 3$. When p_2 and p_3 are consumed, they are returned to the free pool, and let us assume those physical pages are immediately used to hold new data, in our case, p_9 and p_{10}. This leaves us with p_4, p_5, p_8, p_9, p_{10} in memory, and $M_d = 2$, $M_f = 3$. This triggers a read IO, so we reserve two physical pages to hold p_6, p_7, and this triggers a write IO (since $M_f = 1 < \sigma_{write}$). But now we have only one page left for incoming data, which is not enough to receive new data during the IOs. The problem occurred because we delayed the write IO too much: If we knew that a read IO was approaching and would consume 2 pages,

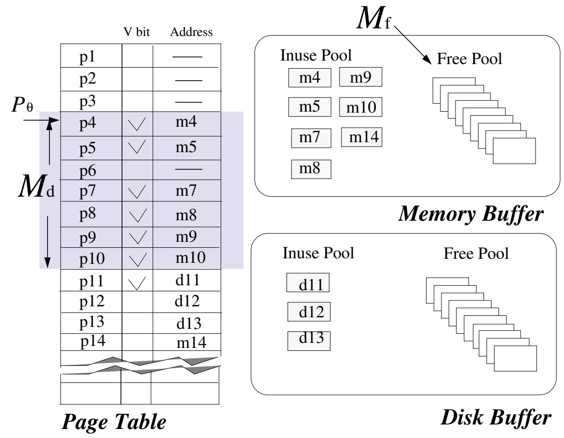


Figure 6: State Variables

we should have started the write earlier, to free up memory.

Our solution is to reserve pages for a read IO in advance, so that we *never* actually request pages for disk reads, and hence we never interfere with the flow of data coming in from the network. Although we could reserve enough memory for read IOs statically, we prefer a dynamic scheme that reserves memory from pages the decoder consumes, and only when a read IO is imminent. To illustrate, let us return to our example when pages $p_2 - p_5$, p_8 are in memory, and p_6, p_7 on disk. When p_2 is consumed by the decoder, we notice that we are approaching the read threshold (p_2 is within $\sigma_{read} + \rho_{read}$ pages from the last sequential page p_5). Thus, the vacated page is *not* returned to the free pool and is reserved for the approaching read IO. Similarly, the vacated p_3 is also reserved. This means that when p_9 arrives, a write IO is triggered. This is earlier than in our previous scenario and gives us enough free space to hold data during the IOs. For a more detailed description of this dynamic reservation policy, please refer to Appendix A.1.

So far we assumed that data arrived in order and that no pages were partially filled. If this is not the case, interesting challenges are introduced. For example, say page p_i is partially filled because a packet is late in arriving. (Page p_2 is in this situation in Figure 5(a).) Assume further that fewer than σ_{write} pages are free, that an IO is initiated to flush pages to disk, and that because of its sequence number, p_i is one of the pages that should be flushed. However, should p_i really be flushed? If we do so, then, when its missing data arrives, we will have to read the page back into memory, update it, and write it out again. Since this can be very expensive, MEDIC tries to keep a page like p_i in memory if at all possible, under the assumption that the missing data will arrive soon in most cases and that it is best to be patient. More specifically, a page like p_i is not written out only if there are ρ_{write} full pages that can be written out when an IO is triggered. If not, all the full pages and some partial pages late in the sequence (for a total of ρ_{write} pages) are flushed to disk.

For M_d accounting purposes, a memory page with missing data is counted as available. For instance, consider the example of Figure 6. Here page p_θ is being decoded, pages p_4, p_5, p_7

– p_{10} are in memory, and page p_{11} is on disk. Page p_6 is missing altogether, but we still say that $M_d = 7$. If page p_6 were partially filled, we would also say $M_d = 7$. The missing p_6 data cannot be read from disk, so reducing M_d to force us to read disk data sooner will not help at all. Thus, it is more accurate to count p_6 as available, and hope that the missing data arrives before the decoder needs it. (Recall that we cannot avoid glitches if data simply does not arrive at the client on time.) Note incidentally that LSF works well with delayed pages. When a page that is early in the memory sequence arrives late, LSF flushes out a page late in the sequence in the next write IO, making room to stage the delayed page when it arrives.

In summary, MEDIC manages memory and disk buffers for supplying data to the decoder and receiving incoming packets. MEDIC keeps track of which pages are available for the decoder in memory, and which are on disk. The algorithm moves pages to disk as necessary, and reads them back later on. It is important to note that MEDIC can only work well if it has enough memory available to handle the input stream and if the control parameters are set properly. This is the topic of the next section.

5 Quantitative Analysis

In this section we analyze MEDIC's use of main memory and derive safe values for the control parameters. Our design goal is to avoid, as much as possible, display glitches due to variability and delays in the input stream. Studies have shown that even just losing 0.1% of data can cause significant degradation in display quality, resulting from inter-frame decoding dependencies [14, 15]. Therefore, in computing our control parameters we take a conservative approach. That is, we assume the worst-case disk latency, as well as peak data consumption and input rates. Of course, we cannot prevent glitches due to data not showing up at the client. However, we do compute how much main memory is required to avoid all glitches due to improper buffering of data within the client.

For our analysis we assume that data arrives at the client in order for now. To safely handle out-of-sequence data requires more main memory to hold partially filled pages than what we estimate below. This is discussed further in Section 5.5.

It is important to note that the *client-side* analysis we present here differs from more "traditional" server-side analyses [10]. In particular, in a server environment, the server has full control of the data generation rate, while at the client, data arrival (from the network) is unpredictable. Since server algorithms have regular "patterns," we know exactly how many IOs are done (and their order) in a given period. At the client, the MEDIC algorithm can generate any sequence of read and write IOs. Thus, unless we are careful, it is hard to predict how many read IOs may occur between two write IOs, and thus how much memory we may need to buffer incoming network data. Another difference is that the client cannot dictate disk layout policies, as is possible on a server. In short, the server schedules events while the client reacts to events; the server has full control in managing its resources while the client may have to reply on off-the-shelf components to manage its files and disk.

5.1 Disk-to-Memory Data Transfer

When M_d reaches the read threshold, σ_{read}, MEDIC allocates ρ_{read} pages and starts a read IO. Let T_r denote the longest time that it takes to complete the IO (we derive T_r in Section 5.4). The maximum amount of data that the decoder can consume in T_r, denoted as $Max_x(T_r)$, can be expressed as

$$Max_x(T_r) = \max_{0 \le \tau \le tt} \sum_{i=\tau}^{\tau+\lceil \frac{T_r}{\delta} \rceil - 1} x_i,$$

where tt denotes the total number of frame periods in x_i. Note that the read threshold must be at least $Max_x(T_r)$, or we may run out of data for the decoder during the T_r period, causing a glitch. In terms of the number of pages, we can thus express the read threshold σ_{read} as:

$$\sigma_{read} \ge \lceil \frac{Max_x(T_r)}{p_{size}} \rceil. \tag{2}$$

To conserve memory, we take the equality in the above expression, yielding

$$\sigma_{read} = \lceil \frac{Max_x(T_r)}{p_{size}} \rceil. \tag{3}$$

Next, we determine a safe value for ρ_{read}. MEDIC only issues a read IO when (1) no read request is pending and (2) M_d has reached σ_{read}. In the worst case, during a T_r time interval, the decoder will consume the σ_{read} data it had available at the beginning of the interval, and the one IO that was issued will provide ρ_{read} pages at the end of the period. To ensure that the decoder has at least σ_{read} pages after this IO, ρ_{read} must be greater than or equal to σ_{read}. To conserve memory we take the equality, and hence

$$\rho_{read} = \sigma_{read}. \tag{4}$$

5.2 Memory-to-Disk Data Transfer

When the number of free pages available, M_f, reaches the write threshold, σ_{write}, MEDIC initiates a write of ρ_{write} pages to the disk to free up memory for future arriving packets. Let T_w denote the longest time the disk takes to complete writing ρ_{write} pages of data (we derive T_w in Section 5.4). The maximum amount of data that can possibly arrive in T_w time at the client can be expressed as

$$Max_y(T_w) = \max_{0 \le \tau \le tt} \sum_{i=\tau}^{\tau+\lceil \frac{T_w}{\delta} \rceil - 1} y_i.$$

If σ_{write} were smaller than this value (in pages), we could run out of free pages to hold incoming data, before the ρ_{write} pages flushed out actually free up. Thus, we set σ_{write} to the smallest safe value, or

$$\sigma_{write} = \lceil \frac{Max_y(T_w)}{p_{size}} \rceil. \tag{5}$$

Next, we determine a safe value for ρ_{write}. Since a write can take as long as T_w to complete, and the maximum amount of data that can arrive in T_w is σ_{write}, MEDIC must write out $\rho_{write} \geq \sigma_{write}$ pages so that at least σ_{write} free pages are available to receive data at the end of the write (which is the earliest time the next write can be started). To conserve memory, we take the equality, and hence

$$\rho_{write} = \sigma_{write}. \tag{6}$$

5.3 Memory Use

MEDIC needs enough main memory to hold the data needed by the decoder, enough free pages for incoming data, and enough pages for read and write IOs. Actually, it is easy to see that, with the parameter settings we have chosen, the following inequality must hold:

$$M_{avail} \geq \sigma_{read} + \rho_{read} + \sigma_{write} + \rho_{write}. \tag{7}$$

That is, $\sigma_{write} + \rho_{write}$ memory is enough to handle all incoming data: while the σ_{write} area fills up, we can write out the ρ_{write} area of the same size. (Keep in mind that with our reservation policy for disk reads, we never use pages in this write area for disk reads.) Similarly, the $\sigma_{read} + \rho_{read}$ area is sufficient for handling any read IOs and any data the decoder may need. Thus, for the rest of our analysis we assume this much memory is available.

5.4 IO Time

In this section, we determine the worst time to complete a read of ρ_{read} pages, T_r, and the worst time to complete a write of ρ_{write} pages, T_w. As a first step we present two theorems that give us the minimum time separation between IOs of the same type.

[Theorem 1]: Suppose the control parameters σ_{read} and ρ_{read} are set according to Equations 3 and 4. Then MEDIC guarantees that (1) the time between the start of two read IOs must be greater than or equal to T_r, and (2) the decoder will have σ_{read} pages of data at the start of every read IO. The proof by induction is presented in Appendix B.1.

[Theorem 2]: Suppose the control parameters σ_{write} and ρ_{write} are set according to Equations 5 and 6. MEDIC guarantees that (1) the time between the start of two write IOs must be greater than or equal to T_w, and (2) the memory pool will have σ_{write} free pages at the start of a write IO. Please see the proof in Appendix B.2.

To compute our T_r and T_w bounds, we assume the worst-case disk latency, including a full rotational delay. We include a full rotational delay because the page size can be much smaller than the track capacity. We assume the disk employs an elevator-like disk scheduling algorithm. In this case, an IO request that involves data blocks scattered on the disk can be serviced by at most two disk arm sweeps. Since the seek function is concave [29, 30], the worst-case total seek overhead occurs when the data blocks are equally separated on the disk surface. In particular, the worst seek overhead for reading ρ_{read} pages occurs when the disk arm must read $\rho_{read}/2$ equally separated pages in each of its two sweeps. Therefore, given that the disk has CYL

cylinders, the worst-case disk latency for reading ρ_{read} pages, denoted by S_r, can be expressed as

$$S_r = \rho_{read} \times \gamma(2 \times CYL/\rho_{read}),$$

where $\gamma(d)$ is a function that computes disk latency (including a full rotational delay) given the seek distance d. (We show a typical $\gamma(d)$ function in the case study of Section 6.) Likewise, the worst disk latency for writing ρ_{write} pages, denoted by S_w, is

$$S_w = \rho_{write} \times \gamma(2 \times CYL/\rho_{write}).$$

Now, suppose that the disk services a read request without interference from any writes. The longest time the disk takes to read the ρ_{read} pages is

$$t_r = S_r + (\rho_{read} \times p_{size})/TR, \tag{8}$$

where TR is the disk transfer rate. Similarly, the longest time to write ρ_{write} pages, without any read interference, is

$$t_w = S_w + (\rho_{write} \times p_{size})/TR. \tag{9}$$

However, read and write IOs can be interleaved. The following theorem provides a bound for the completion time of IOs with interference:

[Theorem 3]: With the control parameters set as described in this section, any IO initiated by MEDIC will complete in time $t_w + t_r$. Please see the proof in Appendix B.3.

Using Theorem 3, we obtain the following inequality for T_r and T_w:

$$T_r = T_w \geq t_w + t_r. \tag{10}$$

5.5 Partially Filled Pages

Recall that for our analysis we have assumed that packets arrive in order, and we have no partially filled pages in memory. Actually, out of order data causes no memory problems as long as it does not generate partially filled pages in memory. (A page that is completely missing from the sequence of pages we are assembling does not consume any space so it is not problematic.) However, a partially filled page "wastes" memory, and this was not accounted for in our analysis. Furthermore, partially filled pages written to disk can generate extra IOs (i.e., data may have to be read from disk to fill it up), and these extra IOs have not been accounted for.

A detailed analysis of how much extra memory is needed to handle partially filled pages is beyond the scope of this paper. Here we simply assume that either the packet size is a multiple of the page size (so we have no partial pages at all), or that enough additional memory is available to hold partial pages, so that no extra IOs are required. We believe that in the latter case a relatively small amount of extra memory will suffice, unless the network is severely damaged. In the rest of the paper we use Equation 7 as our memory bound.

5.6 Solving Control Parameters

Replacing σ_{read} by ρ_{read}, and σ_{write} by ρ_{write}, we can summarize the equations we have derived:

$$\rho_{read} = \lceil \frac{Max_x(T_r)}{p_{size}} \rceil, \; Max_x(T_r) = \max_{0 \leq \tau \leq tt} \sum_{i=\tau}^{\tau + \lceil \frac{T_r}{\delta} \rceil - 1} x_i. \tag{11}$$

$$\rho_{write} = \lceil \frac{Max_y(T_w)}{p_{size}} \rceil, \; Max_y(T_w) = \max_{0 \leq \tau \leq tt} \sum_{i=\tau}^{\tau + \lceil \frac{T_w}{\delta} \rceil - 1} y_i. \tag{12}$$

$$t_r = \rho_{read} \times \gamma(\frac{2 \times CYL}{\rho_{read}}) + \frac{\rho_{read} \times p_{size}}{TR}. \tag{13}$$

$$t_w = \rho_{write} \times \gamma(\frac{2 \times CYL}{\rho_{write}}) + \frac{\rho_{write} \times p_{size}}{TR}. \tag{14}$$

$$T_r = T_w \geq t_w + t_r \tag{15}$$

$$M_{avail} \geq 2 \times (\rho_{read} + \rho_{write}). \tag{16}$$

Note that if we do not have available the complete x_i and y_i functions, we may use bounds that give us the maximum data that can arrive (or can be consumed) per frame. (These bounds can replace x_i and y_i in the computation of $Max_y(T_w)$ and $Max_x(T_r)$ respectively.) Such bounds may be obtainable from the intrinsic limits of the network and the decoding scheme. These bounds can be rather conservative, since as we will see later, relatively small amounts of main memory can handle large network and consumption rates.

Given the page size and disk hardware parameters, we can solve for the control parameters based on an optimization objective. Although several objectives are possible, the following two are among the most interesting:

- Minimizing memory: Given the peak input and consumption rates, find the smallest value of M_{avail} for which we have feasible control parameters.
- Maximizing input bitrate: Given the peak consumption rate and the available memory, find the peak input rate, $Max_y(T_w)$, that the client can support, and find the corresponding control parameters.

Our set of equations can be solved numerically for each optimization scenario. For example, Figure 7 illustrates a simple iterative search procedure for obtaining the minimum memory required and the corresponding control parameters. (The steps to obtain the peak supportable input bitrate are similar.) The iterative procedure of Figure 7 uses a *stride*, α, to increase T_w and T_r as it searches for a feasible solution. The value of α should be set small enough to obtain a tolerable "error." (Here error means that we may estimate a memory size that is slightly larger than absolutely necessary. We used a value of $\alpha = 1$ millisecond in Section 6.) Clearly, there are more efficient procedures for solving these equations, but since this computation is performed off-line, our focus here is simply on illustrating the process.

In Figure 7 we continue until a feasible solution is found, when $T_r \geq t_r + t_w$ (i.e., $d_{new} < 0$). However, we also stop if it is clear there is no feasible solution. This happens when the gap between T_r and $t_r + t_w$ grows from one iteration to the next (i.e., when $d_{old} > d_{new}$).

- Input: α
- Initialization
  - Variables: d_{new}, d_{old}
  - $d_{new}, d_{old} \leftarrow -\infty$
- Execution Steps:

1. $T_r, T_w \leftarrow 0$
2. While $((d_{new} < 0)$ and $(d_{new} > d_{old}))$
   (a) $d_{old} \leftarrow d_{new}$
   (b) $T_r \leftarrow T_r + \alpha$; $T_w \leftarrow T_w + \alpha$
   (c) Compute $Max_x(T_r)$ and $Max_y(T_w)$.
   (d) Compute ρ_{read} and ρ_{write} following Eqs 11 & 12.
   (e) Compute t_r and t_w following Equations 13 and 14.
   (f) $d_{new} = T_r - (t_w + t_r)$
3. If $d_{new} \geq 0$ (feasible solution found)
   (a) Compute $M_{avail} = 2 \times (\rho_{read} + \rho_{write})$.
   (b) Output $\rho_{read}, \sigma_{read}, \rho_{write}, \sigma_{write}$, and M_{avail}.

Figure 7: Steps for Min. M_{avail}

5.7 Initial Playback Delay

At the end of Section 3 we derived ψ, the initial delay before playback starts. The following equation converts the initial delay into the number of pages MEDIC must have received before playback starts.

$$\sigma_{playback} = \sum_{i=1}^{\psi} x_i.$$

In addition, we must ensure that playback does not start until at least σ_{read} pages are ready for the decoder. Thus, our equation is amended to:[2]

$$\sigma_{playback} = \max\{\sum_{i=1}^{\psi} x_i, \sigma_{read}\}. \tag{17}$$

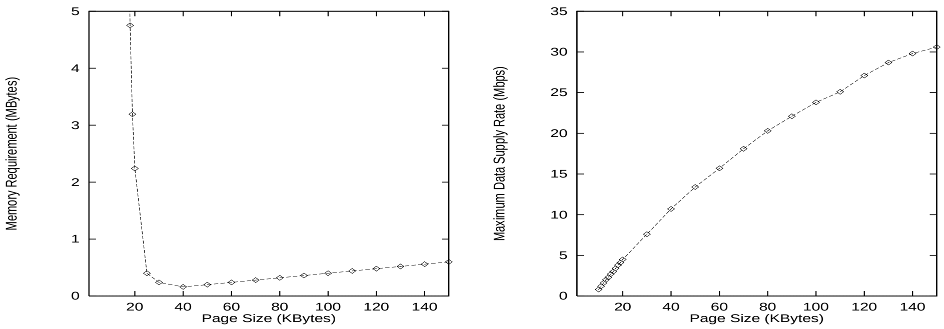
6 Evaluation and Observations

In this section we study MEDIC's performance with an IBM Deskstar disk, a unit designed for desktop computers. Given this hardware, the only parameter that may be varied to tune IO

[2] This equation assumes no partially filled pages. If we expect partial pages, the count must be adjusted so that $\sigma_{playback}$ pages hold the equivalent amount of data.

Parameter Name	*Value*
Disk Capacity	*8.45 GBytes*
Number of cylinders, CYL	*9784*
Min. Transfer Rate TR	*76.2 Mbps*
Max. Transfer Rate TR	*127.4 Mbps*
Full Rotational Latency Time	*11.2 milliseconds*
Min. Seek Time	*2.2 milliseconds*
Max. Seek Time	*15.5 milliseconds*
$\alpha 1$	*2.0 milliseconds*
$\beta 1$	*0.2 milliseconds*
$\alpha 2$	*7.24 milliseconds*
$\beta 2$	*0.000844 milliseconds*

Figure 8: IBM Deskstar DHEA-38451 Disk Parameters



(a) M_{avail} vs. p_{size} (b) Max. Bitrate vs. p_{size}

Figure 9: The Effects of Page Size

performance is the page size p_{size}. Thus, many of the experiments in this section focus on how the page size affects disk utilization, and hence our performance objectives.

Figure 8 lists the parameters for the IBM Deskstar disk. In addition, we assume that the peak data consumption and input rates are both 4 Mbps (a typical MPEG2 bitrate).

For computing the seek overhead we follow closely the model developed in [20, 29], which has been proven to be asymptotically close to the real disks. The seek overhead function is the following concave function:

$$\gamma(d) = \alpha 1 + (\beta 1 \times \sqrt{d}) + 11.2 \text{ if } d < 900$$

$$\gamma(d) = \alpha 2 + (\beta 2 \times d) + 11.2 \text{ if } d \geq 900$$

6.1 Page Size

Figure 9 shows the effect of page size on required memory and on peak input rate. The horizontal axis represents the memory page size up to 150 KBytes. (The increments are one KByte when the page size is less than 20 KBytes, and 10 KBytes when $p_{size} \geq 20$ KBytes. Page sizes beyond 150 KBytes convey no additional information.) Figure 9(a) shows the minimum memory requirement for a peak input rate of 4 Mbps for different page sizes. Similarly, Figure 9(b) shows the maximum data input rates that MEDIC can support given 4 MBytes of main memory. We make three observations from these results.

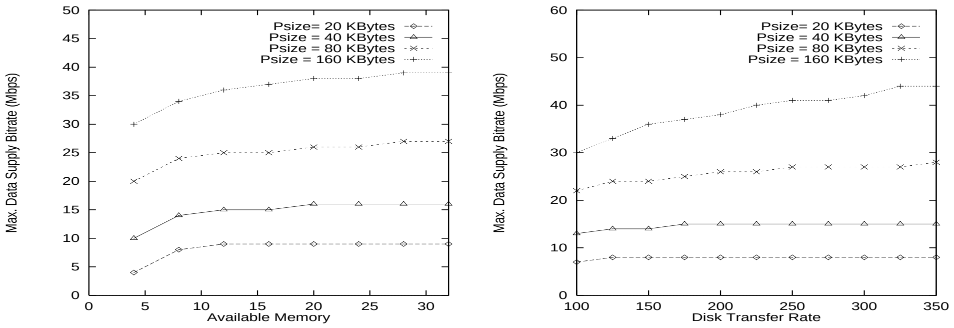
- Not all page sizes yield feasible solutions. Figure 9(a) shows that when $p_{size} < 19$ KBytes, no feasible control parameters exist to support the peak data consumption and input rates of 4 Mbps. Figure 9(b) confirms that the supportable peak rate is 3.8 Mbps when $p_{size} = 18$ KBytes and 4.1 when $p_{size} = 19$ KBytes.
- Figure 9(a) shows that the memory requirement exhibits a sharp knee at $p_{size} = 40$ KBytes. The minimum memory requirement drops drastically as the page size goes up, as p_{size} approaches 40 KBytes. A larger page size decreases the number of inter-block seeks and consequently leads to more efficient IOs and memory savings. However, as soon as the data that arrives during one write IO is able to fit into one single page, increasing page size further only wastes memory (because of internal fragmentation). This explains why the memory requirement goes up when $p_{size} > 40$ KBytes. Thus, we should avoid selecting a larger than necessary page size.
- The maximum data input rate MEDIC can handle goes up with the page size as shown in Figure 9(b). This improvement results from the reduction in the inter-block seek overhead.

To summarize, a large page size can increase disk bandwidth, and may reduce the required memory or increase the peak supported input rate. However, a large page size may not be possible, for instance because the file system has a fixed block size, or because it increases external fragmentation for the disk. An alternative to large pages may be to allocate sets of smaller disk blocks contiguously on disk. The smaller disk and memory pages lead to more effective memory use, and the fact we can write sets of small pages with fewer IOs can improve disk bandwidth. We explore the potential impact of contiguous disk allocation in more detail later on.

6.2 Further Improvements

To further improve the peak rate that MEDIC can support, more memory or a faster disk may seem the right solution. Our evaluation, however, suggests that adding hardware resources does not help greatly. Figure 10(a) shows that additional memory does not significantly improve MEDIC's peak input rate. For instance, when the page size is 20 KBytes, the difference in the peak rate between having 4 and having 32 MBytes of memory is only 4 Mbps. When page size is increased from 20 to 40 KBytes (the memory size fixed at 4 MBytes), the rate goes up by 7 Mbps. The improvements obtained by increasing the page size are much more impressive, as can be seen by the vertical distance between the different page size lines in Figure 10(a).

Figure 10(b) shows the effect of a faster disk transfer rate on the peak input rate. When the page sizes are small, disk latency dominates IO time, and a higher disk transfer rate helps



(a) Max. Bitrate vs. M_{avail} (b) Max. Bitrate vs. TR

Figure 10: The Effect of More Memory or Faster Disk

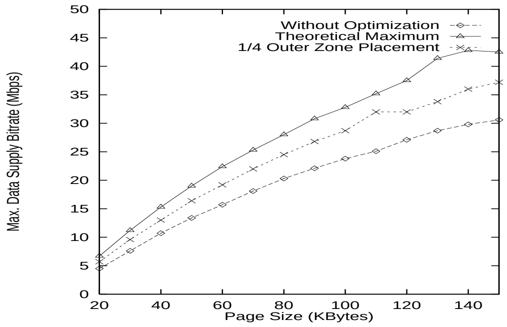


Figure 11: Comparison of the Achievable Bandwidth

little. For example, say the data transfer time takes up only 10% of the IO time. Then reducing the data transfer time by 50% (with a twice as fast disk) reduces the IO time by only 5%, an insignificant improvement. The curves for $p_{size} = 20, 40$ KBytes are thus flat. As the page size grows, the IO becomes more efficient, and the transfer time becomes more dominant in an IO. Now, if we employ a fast disk, the reduction in IO time is noticeable. It is therefore clear that a faster disk is useful only when the page size is large. When the disk is fragmented and its utilization is low, upgrading the disk is merely wasting money!

Of course, the improvements due to a faster disk transfer rate will be more pronounced if the input rate is higher. Clearly, in MEDIC, the disk must be able to keep up with the input stream, else no amount of memory will be sufficient.

As mentioned earlier, another way to improve performance is to enhance disk allocation to reduce the number and the length of seeks. In a perfect situation, if both read and write IOs happen in the same or in adjacent cylinders, the disk latency can be minimized. This, however, is difficult to enforce because read and write rates may vary over time. However, "tricks" such as placing sets of blocks contiguously, or only using the outer tracks of the disk (outer tracks enjoy higher transfer rates in modern disks) may help reduce overheads.

In order to understand the *potential* benefits of such enhancements, we conducted the experiment of Figure 11. The figure shows the maximum supportable input rate under three scenarios. The first scenario is the one we have so far presented in this paper, i.e., the worst-case disk latency. The bottom curve in Figure 11 shows the performance for this scenario (it is the same curve displayed in Figure 9(b).)

In our second scenario we assume the best possible disk latencies. That is, we assume that each inter-page seek travels only one track. (Unless we design for a fixed page size, this is the best we can do.) Such low latencies may not be achievable in practice, but we include them here as a "theoretical" limit. The top curve in Figure 11 shows this best scenario.

Finally, the middle curve in the figure is computed based on an outer track data placement policy. In this scenario, the outer quarter of the tracks is used to hold all data, reducing seek times. The result shows that the outer track data placement policy comes close to the theoretical optimum. Therefore, if we are given the freedom to select the data layout on the client's disk, it is advisable to use the outer tracks for the multimedia cache and the inner tracks for placing conventional data. However, again we observe that having large page size (possibly emulated by contiguous allocation of sets of blocks) is much more beneficial than the data placement techniques. For example, raising page size from $p_{size} = 40$ to $p_{size} = 60$ achieves the same peak input rate as the ideal case at $p_{size} = 40$ KBytes.

6.3 Recommendations

To summarize, to improve IO performance for the client, we recommend the following measures for the local file system:

1. Avoid fragmentation of the local disk, so blocks are allocated more frequently contiguously on disk. This is accomplished by running disk defragmentation software during off-hours.[3]
2. If possible, place the media data only on the outer tracks of the disk.
3. Judiciously adding memory or upgrading to a faster disk, since in many cases increasing these resources will not help.

Finally, it is important to note that MEDIC can support relatively high bit rates with only about 4 to 10 MBytes of memory. We expect these low requirements to hold across many scenarios of interest. Without our integrated memory-disk cache, the memory requirements would be much higher, especially if there is a mismatch between the input and consumption rates. In such a case, memory needs can grow linearly with the length of the presentation (can easily be hundreds of MBytes). Moreover, to assist in server load balancing and support VCR-like functions, the client (e.g., set-top box) may need GBytes of cache space. MEDIC minimizes the main memory requirement and hence minimizes the cost to deploy such intelligent set-top boxes.

[3] Note that as long as MEDIC allocates blocks that are larger than $\max(\rho_{read}, \rho_{write})$, MEDIC does not cause disk fragmentation. For instance, MEDIC can allocate a fixed number of blocks and use them cyclically. If $\rho_{read} > rho_{write}$, a read takes at most one seek and a write at most two (a read always reads a block but a write may cross at most one block boundary). If $\rho_{read} < \rho_{write}$ then a write takes at most one seek and a read at most two. Since the blocks are reused, MEDIC does not inflict disk fragmentation for its future block allocation.

7 Conclusion

We have proposed a client-side adaptive caching scheme that provides a continuous data supply to the decoder, even if the channel has significant delays or errors. Isolating the client from server and network variability and handling the mismatches between data delivery and consumption rates require substantial buffer space, so an integrated memory and disk cache like MEDIC is highly efficient and cost effective. Such client isolation is extremely useful if the end user requires VCR-like functions like pause and rewind and also helpful in assisting in server load balancing as we described in Section 1.

Our analysis of MEDIC showed how the control parameters can be set so that no data is lost due to cache mismanagement and so that the least amount of memory or the largest input data rate can be supported. Our results showed the interaction among the various design parameters and provided guideliness for designing a MEDIC cache. We demonstrated that, contrary to the common belief, using a faster disk may not significantly improve the cache performance. Rather, avoiding disk fragmentation is much more beneficial.

The MEDIC cache can easily be extended to deal with multiple network streams and with playback from a local CD-ROM. For example, one or more servers could send one video base stream, one video enhanced stream, and one audio stream. To handle these multiple streams, we would only have to compute the total combined input and consumption functions or bounds. A stream originating from a local CD-ROM could be simply treated as a presentation that has already been written to the disk cache (the peak data arrival rate would be zero).

In summary, a dynamic, effective, and economical memory-disk cache at the client side complements a good server design. Together the client and server thus provide a complete end-to-end solution for media data delivery.

References

[1] Internet stream protocol specification. *RFC 1819*, August 1995.

[2] H.261 specification. *ITU*, August 1996.

[3] Rtp: A transport protocol for real-time applications. *RFC 1889*, January 1996.

[4] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *Proc. VLDB Conference*, 1989.

[5] R. Abbott and H. Garcia-Molina. Scheduling i/o requests with deadlines: A performance evaluation. *Proc. RTTS Symposium*, 1990.

[6] K. Brown, M. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. *Proc. VLDB Conference*, pages 328–41, 1993.

[7] A. Campbell and G. Coulson. A qos adaptive transport system. *Proceedings of ACM Multimedia*, pages 117–127, Nov. 1996.

[8] M. Carey, R. Jauhari, and M. Livny. Priority in dbms resource scheduling. *Proc. VLDB Conference*, pages 397–410, 1989.

[9] E. Chang and H. Garcia-Molina. Bubbleup - low latency fast-scan for media servers. *Proceedings of the 5th ACM Multimedia Conference*, pages 87–98, November 1997.

[10] E. Chang and H. Garcia-Molina. Effective memory use in a media server. *Proceedings of the 23rd VLDB Conference*, pages 496–505, August 1997.

[11] E. Chang and H. Garcia-Molina. Reducing initial latency in media servers. *IEEE Multimedia*, 4(3):50–61, July-September 1997.

[12] M.-S. Chen, H.-I. Hsiao, C.-S. Li, and P. Yu. Using rotational mirrored declustering for replica placement in a disk-array-based video server. *ACM Multimedia Systems*, December 1997.

[13] M.-S. Chen and D. D. Kandlur. Stream conversion to support interactive video playout. *IEEE Multimedia*, 3(2):51–58, Summer 1996.

[14] I. Dalgic and F. Tobagi. Characterization of quality and traffic for various video encoding schemes and various encoder control schemes. *Stanford Technical Report CSL-TR-96-701*, August 1996.

[15] I. Dalgic and F. Tobagi. Performance evaluation of ethernets and atm networks carrying video traffic. *Stanford Technical Report CSL-TR-96-702*, August 1996.

[16] M. Franklin, M. Carey, and M. Livny. Local disk caching for client-server database systems. *Proc. VLDB Conference*, pages 641–54, August 1993.

[17] S. Ghandeharizadeh, S. Kim, and C. Shahabi. On configuring a single disk continuous media server. *Sigmetrics Performance Evaluation*, 23(1):37–46, May 1995.

[18] J. Howard. Scale and performance in a distributed file system. *ACM TOCS*, 6(1), Feb. 1988.

[19] T. Johnson and A. Zhang. A framework for supporting quality-based presentation of continuous multimedia streams. *Proceedings of the 4th IEEE Conference on Multimedia Computing and Systems*, pages 169–176, June 1996.

[20] D. Kotz, S. B. Toh, and S. Radhakrishnan. A detailed simulation model of the hp 97560 disk drive. *Dartmouth College Technical Report PCS-TR94-220*, 1994.

[21] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *ACM Journal*, January 1973.

[22] F. Moser, A. *Kraiβ*, and W. Klas. L/mrp: A buffer management strategy for interactive continuous data flows in a multimedia dbms. *Proc. VLDB Conference*, pages 275–86, 1995.

[23] S. Murthy and H. Lalgudi. Impact of qos requirement on video coding for atm networks. *Multimedia Systems*, 4:316–27, 1996.

[24] B. Ozden, A. Biliris, R. Rastogi, and A. Silberschatz. A low-cost storage server for movie on demand databases. *Proc. VLDB*, September 1994.

[25] H. Pang, M. Carey, and M. Livny. Managing memory for real-time queries. *ACM Sigmod*, pages 221–32, 1994.

[26] S. Ramanathan and P. Rangan. Adaptive feedback techniques for synchronized multimedia retrieval over integrated networks. *IEEE/ACM Transactions on Networking*, 1(2):246–290, April 1993.

[27] P. V. Rangan and H. M. Vin. Efficient storage techniques for digital continuous multimedia. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):564–573, August 1993.

[28] A. Reddy and J. Wyllie. I/o issues in a multimedia system. *Computer*, 2:69–74, March 1994.

[29] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 2:17–28, 1994.

[30] F. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming raid-a disk array management system for video files. *First ACM Conference on Multimedia*, August 1993.

[31] P. Yu, M.-S. Chen, and D. Kandlur. Grouped sweeping scheduling for DASD-based multimedia storage management. *Multimedia Systems*, 1(1):99–109, January 1993.

Appendix A - MEDIC Specification

We here describe MEDIC formally. After first initializing the state variables, MEDIC is driven by system events. During the playback of a presentation, the following are possible events:

1. The *NewData* event indicates that a network packet has arrived.

   When the *NewData* event occurs, MEDIC checks if the arrived data meet the display deadline. If they do, MEDIC calls procedure LSF (described in Appendix A.1) to update state variables, and perform page replacement if necessary. MEDIC then copies the data onto the memory pages according to the offset and amount of the data in the bit stream. If the playback threshold has been reached and the playback has not started, MEDIC informs the decoder to start decoding and playing back the media data.

2. The *PageConsumed* event indicates that a page of data in the memory buffer has been consumed by the decoder.

   When the *PageConsumed* event occurs, MEDIC calls procedure LSF to update state variables and check if the memory page can be released. If the replenishment threshold, σ_{read}, has been reached (i.e., $M_d \leq \sigma_{read}$) and a data replenishment has not been requested, MEDIC makes the next ρ_{read} virtual pages available to the decoder by calling procedure *ReplenishPages* (described in Appendix A.2).

3. The *IOCompletion* event indicates that an IO has been completed.

   When a read is completed, MEDIC updates the page table entries and releases the disk blocks. When a write is completed, MEDIC updates the page table entries and frees up the memory pages. MEDIC also updates the applicable state variables.

Figure 12 specifies MEDIC formally.

A.1 Procedure LSF

Procedure LSF implements page replacement policy LSF. Procedure LSF is called in two circumstances: 1) when a page is consumed by the decoder, and 2) when new data have arrived.

LSF gives the pages early in the memory sequence higher memory allocation priority. This is implemented through two rules:

- Replacement rule: When pages must be flushed to make room for the incoming packets, LSF flushes out the pages with highest logical sequence number.
- Reservation rule: When a page is consumed, the freed page is reserved for the next read IO if page $p_{\theta+\sigma_{read}+\rho_{read}-1}$ is on disk.

The *replacement rule* is intuitive: we simply do not want to flush out pages that will shortly be needed. To explain the *reservation rule*, we first depict the virtual memory space in Figure 13. LSF treats the area starting from the page that is being decoded, p_θ, to page $p_{\theta+\sigma_{read}+\rho_{read}-1}$, as the *safe zone*. The size of the safe zone is $\sigma_{read} + \rho_{read}$ because when the read threshold, σ_{read}, is reached we want to be sure that we have enough memory to bring in the next ρ_{read} pages. If we are short of memory for these $\sigma_{read} + \rho_{read}$ pages, the data supply to the decoder may be disrupted.

A page that belongs to the safe zone must either already be memory resident or the memory for it must be guaranteed when needed. If a page in the safe zone has had memory, the replacement rule ensures that its memory will not be taken away. If a page in the safe zone is invalid because it does not have a memory page, the replacement rule allocates a memory page

Resource Manager Policy

- Initialization:

  1. Compute the control parameters
     $\{\sigma_{read}, \sigma_{write}, \rho_{read}, \rho_{write}\, \sigma_{playback}\}$
  2. $PlaybackStarted, WriteRequested,$
     $ReadRequested \leftarrow false$
  3. $\{M_d, M_f, \beta_{disk}\} \leftarrow \{0, M_{avail}, 0\}$
  4. $\theta \leftarrow 1$

- Execute the following steps until playback ends:

  1. On *NewData* event
     - Receive the packet
     - If deadline has not passed
       * Execute procedure LSF
       * Copy the data onto the memory pages
       * If $(M_{avail} - M_f + \beta_{disk} > \sigma_{playback})$ & (not PlaybackStarted)
         - Start playback
         - $PlaybackStarted \leftarrow true$
     - Else (the deadline has passed), discard the packet
  2. On *PageConsumed* event
     - Execute procedure LSF
     - If $(M_d \leq \sigma_{read})$ & (not ReadRequested)
       - Execute procedure *ReplenishPages*
       - $ReadRequested \leftarrow true$
  3. On *IOCompletion* event
     - On a read completion
       * Update β_{disk}
       * $M_d \leftarrow M_d + \rho_{read}$
       * $ReadRequested \leftarrow false$
     - On a write completion
       * $\beta_{disk} \leftarrow \beta_{disk} + \rho_{write}$
       * $M_f \leftarrow M_f + \rho_{write}$
       * $WriteRequested \leftarrow false$

Figure 12: MEDIC Specification

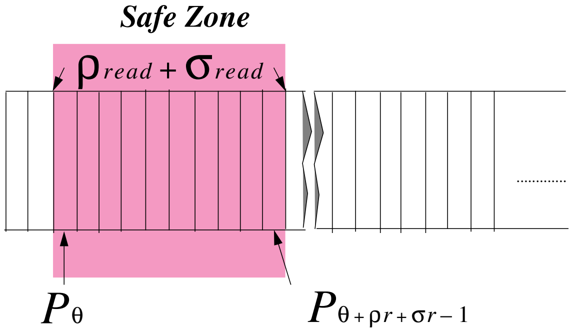


Figure 13: Safe Zone

for it when the page's data arrives. (The replacement rule will flush out a page outside of the safe zone to make room for the page in the safe zone.) The final case is that in which a page in the safe zone is on disk, and the question is when to allocate memory for it.

One solution is allocating memory only when a read IO is initiated. This solution, however, requires more memory resources. For instance, let $M_{avail} = 8$ and all control parameter settings be 2. Let pages p_4, p_5, p_8, p_9, p_{10}, p_{11} be in memory, and p_6, p_7 on disk. At this time, since M_d and M_f are 2, both read and write thresholds are met. MEDIC needs 4 free pages: 2 for the read IO, and 2 for receiving incoming packets while the write of p_{10} and p_{11} is in progress. But since we have only two free pages, we are short two memory pages. If M_{avail} is 10, and the write threshold is set to 4, the IOs can then proceed without problems. However, the memory requirement would be higher than 8 pages, the requirement of the LSF algorithm.

LSF can use 8 memory pages to perform IOs without causing disruptions. To accomplish this, LSF retains its memory page when a page is consumed if the page that is $\sigma_{read} + \rho_{read}$ pages away (from the consumed page) is on disk. The safe zone shifts one page to the right when a page is consumed. If the new page brought into the safe zone is on disk, LSF gives it the page just freed. Following the above example, when pages p_2 and p_3 are consumed, since the pages brought into the safe zone, p_6 and p_7, are on disk, one page is reserved for each. Meanwhile, since two pages have been reserved for p_6 and p_7, pages p_{10} and p_{11} will have been flushed to disk. (Pages later in the sequence yield to the pages earlier in the sequence.) At the time when both read and write are initiated, we have 4 pages (p_4, p_5, p_8, p_9) in memory, 2 pages reserved for bringing in p_6 and p_7, and 2 free pages for receiving incoming packets. Incidentally, to ensure that LSF works, the memory cannot be less than 8 pages in the example we have just shown. Otherwise, a page in the safe zone may be flushed to disk, and the safe zone is no longer safe. Section 5.3 derives the minimum memory bound for LSF.

Let p_i.valid denote whether the i^{th} page has arrived at the client. Let p_i.memory denote the pointer to the physical memory address and p_i.disk denote the pointer to the disk block. Figure 14 documents algorithm LSF.

A.2 Procedure ReplenishPages

As soon as the data in memory runs under the replenishment threshold σ_{read}, MEDIC prepares the next ρ_{read} virtual pages available to the decoder by calling procedure *ReplenishPages*. Note that some of the next ρ_{read} pages of data may be in memory, on disk, or may not yet have arrived. MEDIC relies on the page table to locate the pages and requests disk IOs if necessary. When a page belonging to one of the next ρ_{read} pages is valid in the page table and resides on disk, the page must be read in from the disk. Procedure *ReplenishPages* calls procedure *DoIOs* to read in on-disk pages. For the data pages that have not arrived, MEDIC at this time allocates memory and in the hope that the data will eventually arrive.

A.3 Procedure DoIOs

Procedure *DoIOs* receives IO requests and must complete the IOs before the deadlines. The implementation of procedure *DoIOs* faces some practical restrictions. First, the data on the client's local disk may not be rearranged at will. Some clients may have reasons to lay out the data in a certain way, or may forbid applications to change the data placement on their disks because of security concerns. Therefore, procedure *DoIOs* may not have the freedom to employ any data placement policies. Second, it is impractical to assume one can replace the client's device driver with a new disk scheduler. Procedure *DoIOs* must run at the application level to work with any off-the-shelf commercial disks. Due to these two implementation constraints, procedure *DoIOs* may have to perform IOs through the local file system calls. Performing IOs through the file system may not allow complete flexibility to maximize the disk bandwidth. However, we point out at the end of Section 6 some measures the client can take to improve its own performance.

Procedure *DoIOs* raises an IOCompletion event when the file system calls are complete.

Appendix B - Proofs of Theorems

B.1 Proof of Theorem 1

[Proof] The proof is by induction on the number of the read request. We denote t_j as the time when the j^{th} read is requested.

- Basis (the first read request):

  When the playback starts (the playback does not start until a sufficient number of pages of data have accumulated. See Section 5.7 for details), M_d must be $> \sigma_{read}$. As soon as M_d drops to the read threshold the first read is initiated. At this time, the decoder has σ_{read} pages, and no other reads have been requested. Thus we have proven the basis.

- Inductive Step:

  Now we have to prove the inductive step. We assume that the j^{th} read satisfies the theorem. The theorem states that $t_j \geq t_{j-1} + T_r$, and the number of pages available to the decoder (M_d) at time t_j is σ_{read}. We want to show that $t_{j+1} \geq t_j + T_r$, and $M_d = \sigma_{read}$ at time t_{j+1}.

  1. The j^{th} read started at time t_j and must have been completed by time $t_j + T_r$, according to the definition of T_r. In this T_r time, the decoder can consume only up to σ_{read} pages, according to Equation 3. In other words, the number of pages available to the decoder at time $t_j + T_r$ must be greater than or equal to zero. At the same time, the read must have been completed, and it adds at least σ_{read} pages to M_d. Thus, M_d must have $\geq \sigma_{read}$ pages at time $t_j + T_r$. According to the specification of MEDIC, the next read IO is not initiated until the page count M_d drops to σ_{read}. Thus, the next read IO must happen after time $t_j + T_r$, or $t_{j+1} \geq t_j + T_r$.
  2. Next, we check if the number of pages available to the decoder is σ_{read} at time t_{j+1}. When a read is requested, not all $\rho_{read} = \sigma_{read}$ pages may reside on the disk. We have two cases:

     (a) All σ_{read} pages are on disk at t_j: As we have just discussed, at time $t_j + T_r$, M_d must have greater than or equal to σ_{read} pages. The $j + 1^{th}$ read either starts immediately if $M_d = \sigma_{read}$, or starts later when M_d drops down to σ_{read}. Either way, $M_d = \sigma_{read}$ at time t_{j+1}.

(b) Not all σ_{read} pages are on disk at t_j: If a data page is not found on disk, the data of the page can be in one of the three states: (1) have arrived but have not been written out, (2) have partially arrived but have not been written out, and (3) have not yet arrived at all. If a page is memory resident (in state 1 or 2), the page will not be flushed to disk before it is consumed according to algorithm LSF. If a page is missing (state 3), MEDIC allocates memory and waits for the data for it to arrive eventually. In any rate, $M_d = \sigma_{read}$ at time t_{j+1}.

The $j+1^{th}$ step satisfies the theorem. Therefore, we have proven the theorem by induction. □

B.2 Proof of Theorem 2

[Proof] The proof follows these arguments:

1. A write is initiated only when the number of free pages M_f drops to σ_{write}, according to the specification of MEDIC. These σ_{write} free pages are sufficient to at least stage the arriving packets for the next T_w time when the write is in progress, according to Equation 5. In other words, by T_w time from the time a write is started, the free page count must be greater than or equal to zero, or $M_f \geq 0$.
2. The time it takes to complete writing σ_{write} pages must be less than or equal to T_w. To put it another way, by T_w from the time when the write is requested, the write must have been completed and σ_{write} additional pages have become available. Thus, we have $M_f \geq \sigma_{write}$.
3. If $M_f = \sigma_{write}$ by T_w from the time when the write is requested, a write IO is initiated immediately. If $M_f > \sigma_{write}$, a write IO is not initiated until M_f drops to σ_{write}. Either way, when the next write is started, the $M_f = \sigma_{write}$, and the next write starts at least T_w after the previous one. □

B.3 Proof of Theorem 3

[Proof] We only prove the theorem for read requests. The proof for write requests is symmetric.

When a read request is issued to the disk, another read request cannot be pending according to Theorem 1. The disk is in one of the following two states:

1. The disk is idle: If the disk is idle, the read request is serviced immediately. According to Theorem 1, another read request cannot arrive before the completion of the current read. A write request, however, can arrive when the disk is servicing the read. Since we assume that the disk uses an elevator-like disk scheduling policy, all disk blocks that the write accesses may happen to be in the way and may stall the completion of the read. The longest time the write can stall the read is the longest time to service the write, or t_w. If we add the worst time to complete the read request, t_r, the read must be completed in $t_w + t_r$. Note that, when the read and write are interleaved, each disk sweep can only have more rather than fewer blocks to amortize the seek distance. Therefore, the worst case read completion time can only be shorter than $t_w + t_r$.
2. The disk is busy: If the disk is busy, it must be servicing a write request. According to Theorems 1 and 2, at most one write request can stall the read request. Thus, the longest time it takes for a read to complete is to wait for at most one write to be completed, t_w, plus its own service time, t_r.

Either way, we have shown that a read is guaranteed to be completed in $t_w + t_r$. □

Algorithm LSF

- Initialization: $max \leftarrow 0$ (max records the largest valid page number)

1. If called by PageConsumed event (Reservation Rule):
   - $\theta \leftarrow \theta + 1$
   - $M_d \leftarrow M_d - 1$
   - If ($p_{\theta+\sigma_{read}+\rho_{read}-1}.disk == nil$)
     $p_{\theta-1}.valid \leftarrow false$
     $M_f \leftarrow M_f + 1$
   - Else (Reserve the memory page)
     $p_{\theta+\sigma_{read}+\rho_{read}-1}.valid \leftarrow true$
     $p_{\theta+\sigma_{read}+\rho_{read}-1}.memory \leftarrow p_{\theta-1}.memory$
     Update M_d

2. If called by NewData event (Replacement Rule)
   - For each page i the new data belong to
     If ($i > max$) max $\leftarrow i$
     If ($p_i.valid = false$)
     • Allocate memory page m_i
     • $p_i.memory \leftarrow m_i$; $p_i.valid \leftarrow true$
     • $M_f \leftarrow M_f - 1$
     • Update M_d
   - If ($M_f \leq \sigma_{write}$) & (not WriteRequested)
     $m \leftarrow max$ (m records last page flushed)
     $PageFlushed \leftarrow 0$
     While ($PageFlushed \neq \rho_{write}$)
     • While ($p_m.memory == nil$)
       $m \leftarrow m - 1$
     • Call DoIOs($p_m.memory$)
     • $m \leftarrow m - 1$
     • $PageFlushed \leftarrow PageFlushed + 1$
     Update M_d
     $WriteRequested \leftarrow true$

Figure 14: Algorithm LSF