Management of Space in Hierarchical Storage Systems*

Shahram Ghandeharizadeh, Douglas J. Ierardi, Roger Zimmermann

Department of Computer Science
University of Southern California
Los Angeles, California 90089

December 5, 1994

Abstract

The past decade has witnessed a proliferation of repositories whose workload consists of queries that retrieve information. These repositories provide on-line access to vast amount of data and serve as an integral component of many applications, e.g., library information systems, scientific applications, and the entertainment industry. Their storage subsystems are expected to be hierarchical, consisting of memory, magnetic disk drives, optical disk drives, and tape libraries. The database itself resides permanently on the tape. Objects are swapped onto either the magnetic or optical disk drives on demand, and later deleted when the available space of a device is exhausted. This behavior will generally cause fragmentation of the disk space over a period of time, resulting in a non-contiguous layout of disk–resident objects. As a consequence, the disk is required to reposition its read head multiple times (incurring seek operations) whenever a resident object is retrieved. This may reduce the overall performance of the system.

This study investigates four alternative techniques to manage the available space of mechanical devices in such hierarchical storage systems. Conceptually, these techniques can be categorized according to how they optimize several quantities, including: (1) the fragmentation of disk-resident objects, (2) the amount of wasted space, and (3) adaptation to the evolving access pattern of an application. For each of these alternative strategies, we identify the fundamental factors that impact the performance of the system and develop analytical models that quantify each factor. These models can be employed by a system designer to choose among competing strategies based on the physical characteristics of both the system and the target application.

*This research was supported in part by the National Science Foundation under grants IRI-9110522, IRI-9258362 (NYI award), and CDA-9216321, and a Hewlett-Packard unrestricted cash/equipment gift.

1 Introduction

A recent trend in the area of databases has been an increase in the number of repositories whose primary functionality is to disseminate information. These systems are expected to play a major role in library information systems, scientific applications (e.g., Brookhaven protein repository [BKW$^+$77], the human genome repository [Cou88], etc.), the entertainment industry, health care information systems, knowledge-based systems, etc. These systems exhibit the following characteristics. First, they provide on-line access to vast amount of data. Second, only a small subset of the data is accessed at a given point in time. Third, a major fraction of their workload consists of read-only queries. Fourth, objects managed by these systems are typically large and irregularly structured. Fifth, their applications consume the data at a high rate and almost always exhaust the available disk bandwidth. Hence, they face the traditional I/O bottleneck phenomenon. As an example, consider the following two applications:

- The health care industry envisions the use of image repositories to manage X-rays, PET and MRI scans, along with other patient records. These repositories enable a physician to retrieve and display an image for further analysis. The size of a still image may vary from several kilobytes to hundreds of megabytes (if not gigabytes) depending on whether it is color or black and white, its resolution and level of detail. For example, an uncompressed 2550×3300 pixel Gray-scale image might be 8.4 megabytes in size. The same image in color would be 25.2 megabytes in size. A repository managing thousands of images might be hundreds of gigabytes in size, with only a small fraction of images being accessed frequently (e.g., those corresponding to the patients currently undergoing diagnosis and treatment). Typically, an application will retrieve an image in a sequential manner for display. The faster the image can be retrieved, the sooner it can be displayed (due to the availability of fast CPUs).
- The entertainment industry envisions the use of video repositories to provide the so-called *video-on-demand* service (the ability to display the movie of choice to a client upon request). Video objects are large in size. For example, a two hour uncompressed video clip based on NTSC[1] for "network-quality" video is approximately 40 gigabytes in size. Moreover, it requires a 45 megabits per second (mbps) sustained bandwidth for its continuous display. With a lossy compression technique (MPEG [Gal91]) that reduces the bandwidth requirement of this object to 1.5 mbps, this object is approximately 1.2 gigabytes in size. A repository that contains thousands of such objects is terabytes in size, with only a handful of them (say the 10 to 50 most popular movies) having the highest frequency of access. A client generally retrieves a movie in a sequential manner for display.

The large size of these databases has led to the use of hierarchical storage structures. This is motivated primarily by dollars and sense: Storing terabytes of data using DRAM would be very

[1]The US standard established by the National Television System Committee.

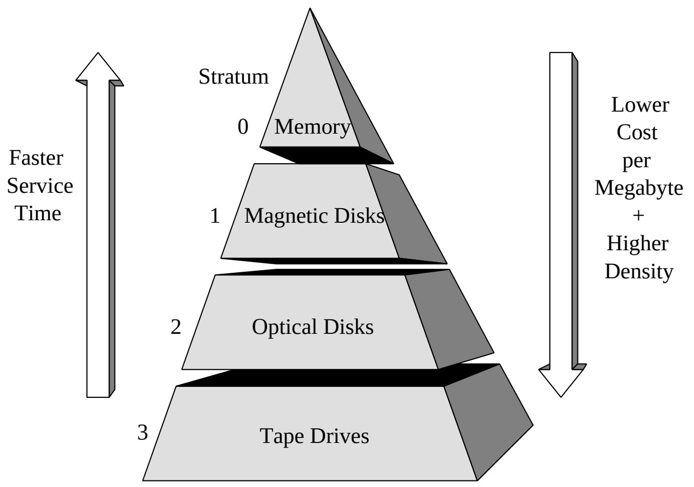

Figure 1: Hierarchical storage system.

expensive. Moreover, it would be wasteful because only a small fraction of the data is referenced at any given instant in time (i.e., due to locality of references). A similar argument applies to other devices, i.e., magnetic disks. The most practical choice would be to employ a combination of fast and slow devices, where the system controls the placement of the data in order to hide the high latency of slow devices using fast devices.

Assume a hierarchical storage structure consisting of random access memory (DRAM), magnetic disk drives, optical disks, and a tape library [CHL93] (see Figure 1). As the different strata of the hierarchy are traversed starting with memory (termed stratum 0), both the density of the medium (the amount of data it can store) and its latency increases, while its cost per megabyte of storage decreases. At the time of this writing, these cost vary from $40/megabyte of DRAM to $0.6/megabyte of disk storage to $0.3/megabyte of optical disk to less than $0.05/megabyte of tape storage. An application referencing an object that is disk resident observes both the average latency time and the delivery rate of a magnetic disk drive (which is superior to that of the tape library). An application would observe the best performance when its working set becomes resident at the highest level of the hierarchy: memory. However, in our assumed environment, the magnetic disk drives are the more likely staging area for this working set due to the large size of objects. Typically memory would be used to stage a small fraction of an object for immediate processing and display. We define the working set [Den68] of an application as a collection of objects that are repeatedly referenced. For example, in existing video stores, a few titles are expected to be accessed frequently and a store maintains several (sometimes many) copies of these titles to satisfy the expected demand. These movies constitute the working set of a database system whose application provides a video–on–demand service.

In general, assuming that the storage structure consists of n strata, we assume that the database resides permanently on stratum $n-1$. For example, Figure 1 shows a system with four strata in which the database resides on stratum 3. Objects are swapped in and out of a device at strata $i < n$, based on their expected future access patterns with the objective of minimizing the frequency of access to the slower devices at higher strata. This objective minimizes the average latency time incurred by requests referencing objects.

At some point during the normal mode of operation, the storage capacity of the device at stratum i will be exhausted. Once an object o_x is referenced, the system may determine that the expected future reference to o_x is such that it should reside on a device at this stratum. In this case, other objects should be swapped out in order to allow o_x to become resident here. However, this migration of objects in and out of strata may cause the available space of the devices to become fragmented, resulting in the non-contiguous layout of its resident objects. Unlike DRAM, optical and magnetic disk drives, and tape drives are mechanical devices. Storing an object non-contiguously would cause the device to reposition its read head when retrieving the object, reducing the overall performance of the device. When it is known that a collection of blocks will be retrieved sequentially, as in the applications considered here, then it is advantageous to store the file contiguously. To demonstrate the significance of this factor, [GR93a] reports that a typical magnetic disk supported by an adequate I/O subsystem can sustain a data rate of 24-40 mbps as long as it is allowed to move its arm monotonously in one direction. With random block accesses scattered across the disk, at saturation point, one would observe a data rate of 3.2 mbps from that same disk. (This analysis assumes a block size of 8 Kilobytes and a service time of 20 millisecond to read a block.) In addition, applications that employ continuous media data types (e.g., audio and video) need to ensure continuous display of each object. To achieve this, such systems must be able to predict the service time of a device, such as a disk drive, in order to schedule the display of one or more objects effectively. When the number of seeks encountered during retrieval of an object is unpredictable, the application may have no choice but to make a conservative estimate on the expected number of seeks to achieve a sufficiently high confidence in its ability to sustain continuous display. As a consequence, memory is wasted, since more data must be staged in memory than is absolutely necessary. (See Appendix A for further details of this application.)

To illustrate the increase in number of seeks as objects are swapped in and out, consider the curve corresponding to Standard in Figure 7. This curve presents the average number of disk seeks required per retrieval of an object as a function of time for a file system that partitions the available disk space into blocks and manages blocks on an individual basis. (Details of the experimental design are outlined in Section 4.2.) The disk starts with a few seeks on behalf of each object and settles with an average of 70 seeks during its later stages of operation. In addition to an increase in the average number of seeks, the variance in this quantity also increases. Of course, the system may employ a re-organization process to ensure the contiguity of blocks which comprise an object. With systems that have a down-time — i.e., become unavailable for some duration of time periodically — the re-organization procedure can be activated as an off-line[2] activity during this period. However, there are applications that cannot tolerate such down-time. For example, health care information systems are expected to provide un-interrupted service 24 hours a day, year round. For systems of this sort, the re-organization procedure must be an on-line process. One may design an effective re-organization process based on the characteristics of the target application. However, it is likely to suffer from the following limitations:

1. The overhead of re-organization can be high if invoked frequently.
2. The re-organization process can respond only when it has detected an undesirable behavior, namely too many seeks. Consequently, the user may observe a lower performance than expected for a while before the re-organization process can remedy the situation.
3. The re-organization process will almost certainly fail in environments where the frequency of access to the objects changes in a manner that the identity of objects resident in a stratum changes frequently. For example, it may happen that, by the time the re-organization process groups the blocks of an object o_x together, the system has already elected to replace o_x with another object that is expected to have a higher number of future references.

One may design a space management technique that ensures a contiguous layout of each object (e.g., REBATE [GI94]). Generally speaking, there is a tradeoff between the amount of contiguity

[2]By *off-line*, we mean that a process is allowed to utilize all of the system resources in order to re-organize the layout of objects as fast as possible. By *on-line*, we mean that the process is allowed to utilize only a fraction of resources to re-organize the layout of objects in order to enable the system to continue to provide service to user requests. In the latter case, the users may observe a degradation in system performance.

guaranteed for the layout of each object on a device at stratum i and the amount of wasted space on that device. For example, a technique that ensures the contiguous layout of each object on a magnetic disk may waste substantial amount of disk space. This tradeoff might be worthwhile if the working set of the target application can become resident on the magnetic disks. It would not be worthwhile if the penalty incurred due to an increasing number of references to slower devices at lower strata outweighs the benefits of eliminating disk seeks.

The contributions of this paper are two-fold. First, it employs the design of the UNIX Fast File System [MJLF84] (termed Standard) to describe the design of three new space management policies: Dynamic, REBATE [GI94], and EVEREST. While Standard packs objects onto the disk without ensuring the contiguity of each object, both Dynamic and REBATE strive to ensure the contiguous layout of each object. EVEREST, on the other hand, strikes a compromise between the two conflicting goals (contiguous layout versus wasted space) by approximating a contiguous layout of each object. In a dynamic environment where the frequency of access to the objects evolves over time, the design of Standard, Dynamic, and REBATE can benefit from a re-organization process that detects and eliminates an undesirable side effect:

- Standard benefits because the re-organization process can ensure a contiguous layout of each object once the system has detected too many seeks per request.
- Dynamic benefits because the re-organization process detects and eliminates its wasted space.
- REBATE benefits because the re-organization process maximizes the utilization of space by detecting and reallocating space occupied by the infrequently accessed objects.

EVEREST is a preventive technique that avoids these undesirable side effects. To the best of our knowledge, the design of EVEREST is novel and has neither been proposed nor investigated to this date.

Second, this study identifies the fundamental factors that impact the average service time of the system using alternative space management policies and models them analytically. These models were verified using a simulation study. They quantify the amount of useful work (transfer of data) and wasteful work (seeks, preventive operations, re-organization, access to slower devices due to wasted space) attributed to a design. The models are independent of those strategies described in this paper and can be employed to evaluate other space management techniques. Thus, they can

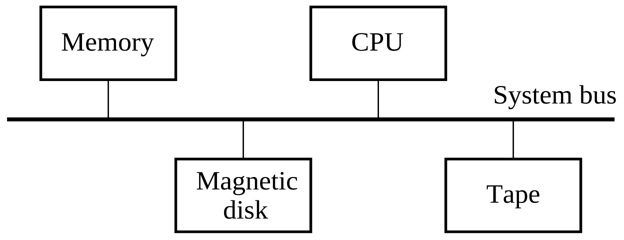

Figure 2: Architecture.

be employed by a system designer to quantify the tradeoff of one technique relative to another with respect to a target application and hardware platform.

The rest of this paper is organized as follows. Second 2 describes our target hardware platform. In Section 3, we describe the four alternative space management techniques using this platform. Section 4 demonstrates the tradeoff associated with these techniques using a simulation study. In Section 5, we develop analytical models that quantify the factors that impact the average service time of a system with alternative strategies. Our conclusions are contained in Section 6.

2 Target Environment

In order to focus on alternative techniques to manage the space of a mechanical device, this study makes the following simplifying assumptions:

1. The environment consists of 3 strata: memory, disk, and tape library. The service time of retrieving an object from tape is significantly higher than that from magnetic disk.
2. The database resides permanently on the tape. The magnetic disk is used as a temporary staging area for the frequently accessed objects in order to minimize the number of references to the tape.
3. All devices are visible to the user via memory (see Figure 2). The memory is used as a temporary staging area either to service a pending request or to transfer an object from tape to the magnetic disk drive.
4. The system accumulates statistics on the frequency of access (heat [CABK88]) to the objects as it services requests for the users. It employs this past history to predict the future number of references to an object.
5. Each referenced object is retrieved sequentially and in its entirety.
6. Either all or none of an object is resident at a stratum; the system does not maintain a portion of an object resident on a stratum. This assumption is justified in the following paragraphs.

With the assumed architecture, the time required to read an object o_x from a device is a function of the size of o_x, the transfer rate of the device, and the number of seeks incurred when reading o_x. The time to perform a seek may include: 1) the time required for the read-head to travel to the appropriate location containing the referenced data, 2) rotational latency time, and 3) the time required to change the physical medium (when necessary). The number of accesses to a device on behalf of an object depends on the frequency of access to that object (its heat).

Once object o_x is referenced, if it is not disk resident and there is sufficient space to store o_x then o_x is rendered disk resident. If the disk drive has insufficient space to store o_x then the system must determine if it should delete one or more objects (victims) from this device in favor of o_x. Generally speaking, the following policy is employed for object replacement. The system determines a collection of least frequently accessed objects (say k of them) whose total size exceeds the size of o_x. If the total heat of these objects ($\sum_{j=1}^{k} heat(o_j)$) is lower than $heat(o_x)$ then the system deletes these k objects in favor of o_x. As described in Section 3, this general replacement policy cannot be enforced with all space management techniques (in particular REBATE and Dynamic). In those cases, we describe its necessary extensions.

An alternative to the assignment imposed by assumption 6 might be to stripe an object across the different strata such that each stratum performs its fair share of the overall imposed work when a request references this object. In the following paragraph, we describe this paradigm and its limitations. These limitations justify assumption 6.

With the striping paradigm, each object is striped into $n-1$ fragments with each fragment assigned to a device at stratum $i = 1, \ldots, n-1$ (no fragments are assigned to memory). In order to void the situation in which one device is waiting for another while requests wait in a queue, the system can choose appropriate sizes for different fragments of each object so that the service time of each device at stratum $i = 1, \ldots, n-1$ is almost identical. Every time the object is referenced, devices at all strata are activated, each for the same amount of time. Hence all devices contribute an equal share to the work performed for each request. To illustrate, assume that the rate of data delivery is t for tape, and $4t$ for magnetic disk. Moreover, assume that this delivery rate is computed by considering the overhead of initial and subsequent seeks attributed to retrieval of an object from a device. With these parameters, this paradigm assigns $\frac{4}{5}$ of o_x to the magnetic disk, and $\frac{1}{5}$ of o_x to the tape. Once o_x is referenced, both devices are activated simultaneously with each completing its retrieval of the fragment of o_x at approximately the same time.

This paradigm suffers from the following limitations. First, for each object o_x, it requires the size of o_x's disk-resident fragment to be larger than the fragment that is resident on the tape, placing larger fragments on devices that have a smaller storage capacity. If all objects are required to have their disk–resident fragments physically present on the disk drive, then the amount of required disk storage would be larger than that of the tape, resulting in a high storage cost. One may reduce the amount of required disk storage (in order to reduce cost) by rendering a subset of objects tape resident in their entirety. Once such an object (say o_x) is referenced, the system employs the tape to retrieve o_x, without the participation of the magnetic disk. During this time, the system may service other requests by retrieving their disk resident fragments. However, should these requests require access to tape resident fragments of their referenced objects then, in effect, the tape has fallen behind and become a bottleneck for the entire system; at some point, the memory (as a temporary staging area for these other objects) will be exhausted, and the disk will sit idle and wait for the tape to catch up.

A second limitation of this approach is its requirement that the different devices must be synchronized so that they complete servicing requests simultaneously. This involves a computation of the delivery rate of a device, perhaps in the presence of a variable number of seeks. This synchronization avoids the scenario in which one device waits for another in the presence of pending requests. Such synchrony is difficult to achieve even in an environment that consists of homogeneous devices, such as multiple magnetic disk drives [PGK88, Gib92]. It becomes more challenging in a heterogeneous system, where each device exhibits its own unique physical characteristics. Due to these limitations, we elected to eliminate striping from further consideration for the remainder of this paper.

3 Four Alternative Space Management Techniques

This section presents four alternative space management techniques: Standard, Dynamic, EVEREST, and REBATE. Standard refers to the most common organization of disk space in current operating systems. We have elected to use the UNIX Fast File System [MJLF84], termed

	Contiguous Layout	May require Re-organization	Wastes Space
Standard	NO	YES	NO
Dynamic	YES	YES	YES
EVEREST	NO	NO	NO
REBATE	YES	YES	YES

Table 1: Characteristics of alternative space management techniques.

UNIX FFS, to represent this class of models. Hence, for the remainder of this paper, Standard = UNIX FFS. While Dynamic and EVEREST are two different algorithms, each can be viewed as an extension to the Standard model. REBATE, however, is a more radical departure that partitions the available disk space into regions, where each region manages a unique collection of similarly sized objects. (A region is not equivalent to a cylinder group, as described by UNIX FFS.)

Both Dynamic and REBATE ensure a contiguous layout of each object. Moreover, both techniques illustrate the benefits and difficulties involved in providing such a guarantee while maintaining sufficiently high utilization of the available space. The design of Dynamic, for example, demonstrates that a "smart" algorithm for ensuring contiguity of objects is both difficult to implement and computationally expensive to support. In addition, it wastes space. REBATE, on the other hand, attempts to simplify the problem by partitioning the available disk space into regions. Within each region, the space is partitioned into fixed-sized frames that are shared by the objects corresponding to that region. In general, however, the use of frames of fixed size will increase the amount of wasted space, and the partitioning of resources makes the technique sensitive to changes in the heat of objects. This sensitivity to changing heats motivates the introduction of a re-organization process that detects these changes and adjusts the amount of space allocated to each region and/or the sets of objects managed by each region.

EVEREST, on the other hand, does not ensure a contiguous layout of each object. Instead, it approximates a contiguous layout by representing an object as small collection of chunks. Each chunk consists of a variable number of contiguous blocks. However, the number of chunks per object and the number of blocks per chunk are a fixed function of the size of an object and configuration parameters. Moreover, the number of chunks is small, bounded logarithmically in the object's size. In contrast to the other strategies, EVEREST is preventive (rather than detective) in its management of space fragmentation. Its advantages include: 1) ease of implementation, 2) a minimal amount of wasted space (comparable to the Standard, in this respect), and 3) no need for an auxiliary re-organization technique. Moreover, the basic parameters of the EVEREST scheme can serve to "tune" the performance of the system, in trading-off time spent in its preventive maintenance and the time attributed to seeks between chunks of resident objects.

We describe each technique in turn, starting with Standard.

3.1 Standard

Traditionally, file systems have provided a device-independent storage service to their clients. They were not targeted to manage the available space of a hierarchical storage structure. However, they serve as an ideal foundation to describe the techniques proposed in this study. We use the Unix Fast File System (UNIX FFS) as a representative of the traditional file systems. We could not justify the use of Sprite-LFS [RO92] in this role (and its detailed design) because it is an extended version of UNIX-FFS designed to enhance the performance of file system for small writes [RO92]; conversely, our target environment assumes a workload consisting of large sequential reads and writes. Similarly, we have avoided file systems that support extent-based allocation (e.g., WiSS [CDKK85]) because their design targets files that are allowed to grow and shrink dynamically; the objects in our assumed environment are static in size.

With UNIX-FFS, the size of a block for device i determines the unit of transfer between this device and the memory. With objects (files) that are retrieved in a sequential manner, the utilization of a device is enhanced with larger block sizes because the device spends more of its time transfering data (performing useful work) instead of repositioning its read head (wasteful, or at least potentially avoidable, work). For example, with UNIX FFS that supports small files, the performance of a magnetic disk drive was improved by a factor of more than two by changing the block size of the file system from 512 to 1024 bytes [MJLF84]. A disadvantage of using large block sizes is internal fragmentation of space allocated to a block: an object consists of several blocks with its last block remaining partially empty. UNIX FFS minimizes this waste of space as follows. It divides a single block into m *fragments*; the value of m is determined at system configuration time. Physically, a file is represented as l blocks and at most $m-1$ fragments. Once a file grows to consist of m

fragments, UNIX FFS restructures the space to form a contiguous block from these fragments.

The advantages of this technique include: 1) its simplicity, 2) its ready availability from the commercial arena, 3) its enhancement of the utilization of space by minimizing waste, and 4) its flexibility: it can employ the general replacement policy that was outlined in Section 2 to respond to changing patterns of access to the objects. A limitation of this technique, however, is its inability to ensure contiguous layout of the blocks of an object on the surface of a device. As the system continues operation and objects are swapped in and out, it will scatter the blocks of a newly materialized object across the surface of the device. This motivates the adoption of a re-organization procedure that will groups the blocks of each object together to ensure its contiguity. Section 1 sketches the drawbacks inherent in such a procedure.

3.2 Dynamic

The method that we term Dynamic is an extension of Standard (Section 3.1) that attempts to guarantee the contiguity of all disk-resident objects. Similar to Standard, the available disk space is partitioned into blocks of a fixed size. However, whenever an object is rendered disk resident, Dynamic requires that the sequence of blocks allocated to that object be physically adjacent. The goal of the object replacement criterion is similar to that described in Section 1, namely to maximize the workload of the device, or the total heat contributed by the collection of disk-resident objects. However, the way that Dynamic strives to achieve this objective differs in the following way.

Let o_x be an object requiring b blocks, and assume that o_x is not disk resident. The replacement policy considers all possible contiguous placements of o_x on the disk. If there is some free region that contains b free blocks, then o_x can be made disk resident in this region, and the workload of the set of disk resident objects increases. On the other hand, if no such free region exists, then it must be the case that every sequence of b contiguous blocks contains all or part of some other resident objects. To be specific, let us fix some sequence of b blocks. Assume that these blocks contain all or part of objects $o_i, \ldots, o_j$, together with zero or more free blocks. If Dynamic were to make o_x resident in these blocks, the disk-resident copies of these objects would be destroyed, in whole or in part. However, since we have assumed that no objects may reside partially on the disk, whenever a single block occupied by a resident object is overwritten then this object is destroyed in its entirety. To determine how the workload might change if o_x is made resident in these blocks, we would like to quantify the amount of work contributed by the current configuration, and compare that to the work expected from the proposed change. To do this we define work as follows.

Definition 3.1 *If o_x is an object, then* $\mathrm{work}(o_x) = \mathrm{heat}(o_x) \times \mathrm{size}(o_x)$.

The definition captures the idea: that part of the disk's workload that may be attributed to requests for object o_x is not merely a function of its heat, but also depends on the amount of time used by the disk to service these requests. This in turn depends upon the object's size. During any period of time during which the objects' heats remain fixed, one expects that this time will be proportional to $\mathrm{work}(o_x)$, for each o_x that is actually disk-resident, if we neglect the initial seek for each access to o_x.

To illustrate why work, rather than heat alone, is required by Dynamic's replacement policy, consider the following example. An object o_x of heat $\frac{1}{10}$ requires 100 blocks to become disk resident. On the disk, there is a region of 100 contiguous blocks in which 90 are free and 10 are occupied by an object o_y of heat $\frac{1}{5}$. On the one hand, we can expect that object o_y will receive twice as many requests as object o_x. On the other hand, suppose that the time required to service a single request for o_y is t (neglecting the initial seek). Then each request for o_x requires $10t$ time. Based on these heats, we can expect that about one in ten requests will reference o_x and one in five will access o_y. So over a sufficiently long sequence of requests, one expects that

$$\frac{\text{time servicing requests for } o_x}{\text{time servicing requests for } o_y} = \frac{\frac{1}{10}10t}{\frac{1}{5}t} = \frac{\mathrm{work}(o_x)}{\mathrm{work}(o_y)} = \frac{5}{1} .$$

Materializing o_x in this region will thus increase the expected workload of the disk.

Dynamic's replacement policy may be stated succinctly as follows. On each request for an object (o_x) that is not disk-resident, Dynamic considers all sequences of blocks where o_x may be placed. For each possible placement, it evaluates the expected change in the workload of the disk. If materialization of o_x will increase this quantity, Dynamic stores o_x in the region that maximizes the workload of the disk. Otherwise, o_x is not materialized.

When Dynamic considers placing o_x in a sequence of b blocks, it first evaluates the work contributed by the current residents $(o_i, \ldots, o_j)$ of those blocks. We define this quantity,

$$\sum_{k=i}^{j} \text{work}(o_k) \ ,$$

to be the work associated with these blocks. Rendering o_x disk resident by overwriting these objects would: 1) increase the workload of the disk by work(o_x), and 2) reduce the workload by the current work associated with objects $o_i, \ldots, o_j$. Hence the expected change in the workload of the disk will be

$$\text{work}(o_x) - \sum_{k=i}^{j} \text{work}(o_k) \ . \tag{1}$$

If (1) is positive then Dynamic materializes o_x as it increases the workload of the disk.

The algorithm that Dynamic uses to determine when and where to materialize an object o_x is a straightforward scan of the disk — or rather, a memory-resident data structure that records the layout of the current disk-resident population. To illustrate, assume that o_x needs 100 blocks to become disk resident. In order to maximize the device utilization, Dynamic must find the 100 contiguous blocks on the device that contribute the least to the device workload. Conceptually, this can be achieved by placing a *window* of 100 blocks at one end of the device, and calculating the total workload of all objects that can be seen through this window. The window is then slid down the length of the disk. Every time that the set of objects visible through this window changes, the visible workload is recalculated, and the overall minimum value m is recorded. After the entire disk is scanned, m is compared to work(o_x), and if work(o_x) $> m$, o_x is materialized in that sequence of blocks with associated workload m. Otherwise, o_x is not materialized on the disk.

The actual calculation can be simplified somewhat by keeping an appropriate memory–resident image of the disk's organization. For this, we employ a list of intervals. Each interval corresponds either (1) to some sequence of blocks occupied by a single object, or (2) to a maximal contiguous sequence of free blocks. All intervals are annotated with their size and their resident object (when the blocks are not free). When a request is made for an object o_x requiring b blocks, Dynamic begins by gathering intervals from the head of the list until at least b blocks have been accumulated. Say this window consists of intervals $I_1, \ldots, I_j$. The total workload of the objects represented among these intervals is recorded. Then, to slide the window to its next interval, the first interval I_1 is omitted, zero or more of the intervals $I_{j+1}, I_{j+2}, \ldots$ are added to the window, until it again contains at least b blocks. The process is repeated across the entire list, while retaining a pointer to the window of minimum workload. It is easy to see that the entire algorithm is linear in the number of disk resident objects d, since the number of intervals (free and occupied) is no more than $2d + 1$, and each interval is added to and removed from the window at most once.

The advantage of this procedure is its ability to guarantee contiguous layout of objects. In addition, similar to Standard, it always uses the most up-to-date heat information in making decisions concerning disk-residency, so its disk configuration is adaptive and responds to changing access patterns. It uses the heat statistics to "pack" the hottest objects onto the disk. Colder objects are removed from the disk when it is found that hotter objects can increase the disk's workload. But each of these decisions must be made in a "greedy", local manner, by considering objects as they are requested. The decisions are further constrained by the current organization of the disk, since Dynamic does not change the layout of those objects that remain disk resident. (More global re-organization of this sort may be effected by an auxiliary re-organization policy.)

Nevertheless, Dynamic can suffer from the following limitations. First, it will almost certainly waste disk space because it does not permit the discontinuous layout of an object. Smaller cold or free sequences of blocks can become temporarily unusable when sandwiched between two hot objects. In effect, the method is restricted in its later placement of data by the current layout, which in turn can evolve in an unpredictable manner. Moreover, Dynamic optimizes the workload of the disk by considering only a local (greedy) perspective. Hence, it may perform wasteful work. For example, it may render an object o_x disk resident, only to overwrite it with a hotter object soon thereafter. Of course, as in the case of Standard, Dynamic can also be augmented with a re-organization scheme that attempts to optimize the layout of disk-resident objects from a more global perspective. Such a re-organization process would be subject to the same limitations as outlined in Section 1. Finally, we note that the algorithm discussed above (for determining whether an object should be materialized and where it should be placed) although linear in the number of disk resident objects, may be time-consuming when the number of objects is large. This may add a significant computational overhead to every request.

3.3 EVEREST

EVEREST is an extension of Standard designed to approximate a contiguous layout of each object

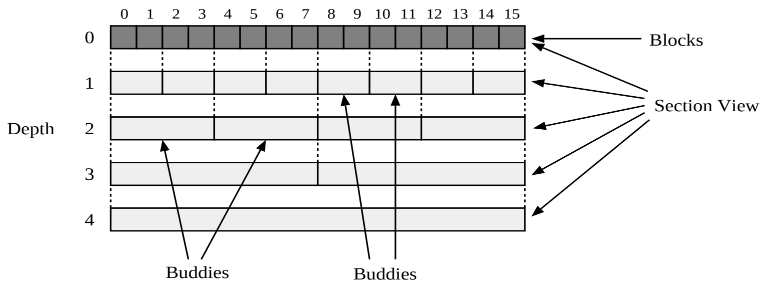

Figure 3: Physical division of disk space into blocks and the corresponding logical view of the sections with an example base of $B = 2$.

on the disk drive. Its basic unit of allocation is a block, also termed *sections* of height 0. Within the EVEREST scheme, these blocks can be combined in a tree-like fashion to form larger, contiguous sections. As illustrated in Figure 3, only sections of size(block) $\times B^i$ (for $i \geq 0$) are valid, where the base B is a system configuration parameter. If a section consists of B^i blocks then i is said to be the height of the section. In general, B height i sections (physically adjacent) might be combined to construct a height $i+1$ section.

To illustrate, the disk in Figure 3 consists of 16 blocks. The system is configured with $B = 2$. Thus, the size of a section may vary from 1, 2, 4, 8, up to 16 blocks. In essence, a binary tree is imposed upon the sequence of blocks. The maximum height, given by[3] $N = \lceil \log_B(\lfloor \frac{Capacity}{\text{size(block)}} \rfloor) \rceil$, is 4. With this organization imposed upon the device, sections of height $i \geq 0$ cannot start at just any block number, but only at offsets that are multiples of B^i. This restriction ensures that any section, with the exception of the one at height N, has a total of $B - 1$ adjacent *buddy* sections of the same size at all times. With the base 2 organization of Figure 3, each block has one buddy. This property of the hierarchy of sections is used when objects are allocated, as described below in Section 3.3.2.

3.3.1 Organization and Management of the Free List

With EVEREST, a portion of the available disk space is allocated to objects. The remainder, should any exist, is free. The sections that constitute the available space are handled by a memory-resident *free list*. This free list is actually maintained as a sequence of lists, one for each section

[3]To simplify the discussion, assume that the total number of blocks is a power of B. The general case can be handled similarly and is described in Section 3.3.4

height. The information about an unused section of height i is enqueued in the list that handles sections of that height. In order to simplify object allocation, the following *bounded list length property* is always maintained:

Property 3.1

For each height $i = 0, \ldots, N$, at most $B - 1$ free sections of i are allowed.

Informally, the above property implies that whenever there exists sufficient free space at the free list of height i, EVEREST *must* compact these free sections into sections of a larger height[4].

3.3.2 Allocation of an Object

Property 3.1 allows for straightforward object materialization. The first step is to check, whether the total number of blocks in all the sections on the free list is either greater than or equal to the number of blocks (denoted no-of-blocks(o_x)) that the new object o_x requires. If this is not the case, one or more victim objects are elected and deleted. (The procedure for selecting a victim is the same as that described in Section 2. The deletion of a victim object is described further in Section 3.3.3 below.) Assuming at this point that there is enough free space available, o_x is divided into its corresponding sections according to the following scheme. First, the number m = no-of-blocks(o_x) is converted to base B. For example, if $B = 2$, and no-of-blocks(o_x) = 13_{10} then its binary representation is 1101_2. The full representation of such a converted number is $m = d_{j-1} \times B^{j-1} + \ldots + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$. In our example, the number 1101_2 can be written as $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. In general, for every digit d_i that is non-zero, d_i sections are allocated from height i of the free list on behalf of o_x. In our example, o_x requires 1 section from height 0, no sections from height 1, 1 section from height 2, and 1 section from height 3.

For each object, the number k of contiguous pieces is equal to the number of one's in the binary representation of m, or with a general base B, $k = \sum_{i=0}^{j} d_i$ (where j is the total number of digits). Note that k is always bounded by $B \lceil \log_B m \rceil$. For any object, k defines the maximum number

[4]A lazy variant of this scheme would allow these lists to grow longer and do compaction upon demand, *i.e.* when large contiguous blocks are required. This would be complicated as a variety of choices might exist when merging blocks. This would require the system to employ heuristic techniques to guide the search space of this merging process. However, to simplify the description we focus on an implementation that observes the invariant described above.

of disk seeks required to retrieve that object. (The minimum is 1 if all k sections are physically adjacent.) A complication arises when no section at the right height exists. For example, suppose that a section of size B^i is required, but the smallest section larger than B^i on the free list is of size B^j ($j > i$). In this case, the section of size B^j can be split into B sections of size B^{j-1}. If $j - 1 = i$, then $B - 1$ of these are enqueued on the list of height i and the remainder is allocated. However, if $j - 1 > i$ then $B - 1$ of these sections are again enqueued at level $j - 1$, and the splitting procedure is repeated on the remaining section. It is easy to see that, whenever the total amount of free space on these lists is sufficient to accommodate the object, then for each section that the object occupies, there is always a section of the appropriate size, or larger, on the list. The splitting procedure sketched above will guarantee that the appropriate number of sections, each of the appropriate size, will be allocated, and that Property 3.1 is never violated.

The design of EVEREST is related to the buddy system proposed in [Kno65, LD91] for an efficient main memory storage allocator (DRAM). The difference is that EVEREST satisfies a request for b blocks by allocating a number of sections such that their total number of blocks equals b. The storage allocator algorithm, on the other hand, will allocate *one* section that is rounded up to $2^{\lceil lg\, b \rceil}$ blocks, resulting in fragmentation and motivating the need for either a re-organization process or a garbage collector [GR93b].

3.3.3 Deallocation of an Object

When the system elects that an object must be materialized and there is insufficient free space, then one or more victims are removed from the disk. Reclaiming the space of a victim requires two steps for each of its sections. First, the section must be appended to the free list at the appropriate height. The second step is to ensure that Property 3.1 is not violated. Therefore, whenever a section is enqueued in the free list at height i and the number of sections at that height is equal to or greater than B, then B sections must be combined into one section at height $i + 1$. If the list at $i + 1$ now violates Property 3.1, then once again space must be compacted and moved to section $i + 2$. This procedure might be repeated several times. It terminates when the length of the list for a higher height is less than B.

Compaction of B free sections into a larger section is simple when the sections are all adjacent

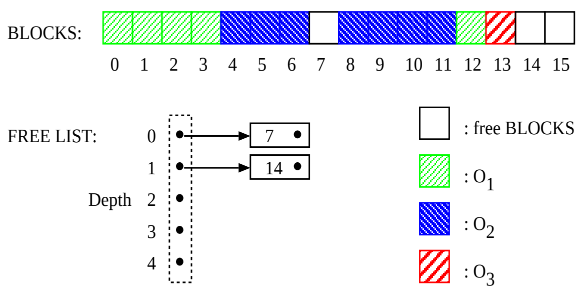

Figure 4a: Two sections are on the free list already (7 and 14) and object o_3 is deallocated.

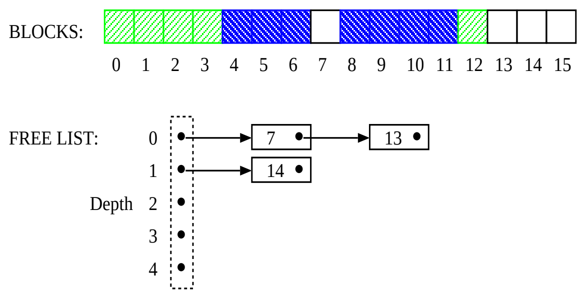

Figure 4b: Sections 7 and 13 should be combined, however they are not contiguous.

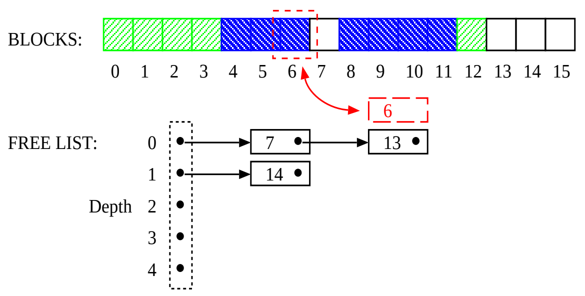

Figure 4c: The buddy of section 7 is 6. Data must move from 6 to 13.

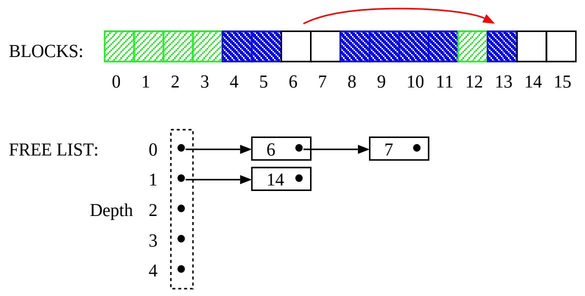

Figure 4d: Sections 6 and 7 are contiguous and can be combined.

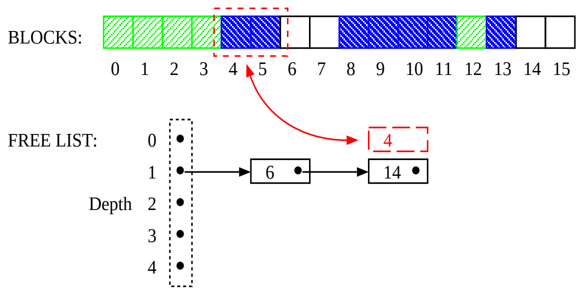

Figure 4e: The buddy of section 6 is 4. Data must move from (4,5) to (14,15).

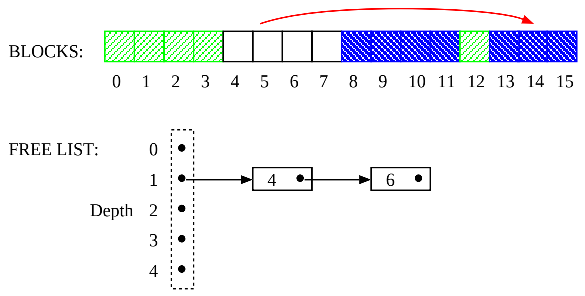

Figure 4f: Sections 4 and 6 are now adjacent and can be combined.

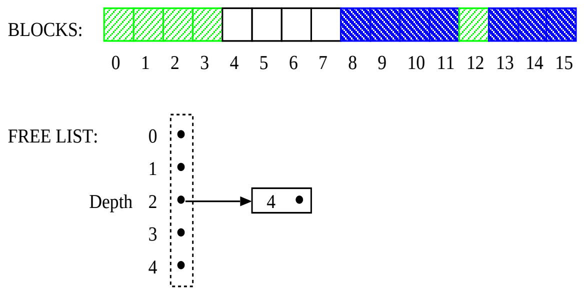

Figure 4g: The final view of the disk and the free list after removal of o_3.

Figure 4: Deallocation of an object. The example sequence shows the removal of object o_3 from the initial disk resident object set $\{o_1, o_2, o_3\}$. Base two, $B = 2$.

to each other; in this case, the combined space is already contiguous. Otherwise, the system might be forced to exchange one occupied section of an object with one on the free list in order to ensure contiguity of an appropriate sequence of B sections at the same height. The following algorithm achieves space-contiguity among B free sections at height i.

1. Check if there are at least B sections for height i on the free list. If not, stop.
2. Select the first section (denoted s_j) and record its block-number (i.e., the offset on the disk drive). The goal is to free $B - 1$ sections physically adjacent to s_j.
3. Calculate the block-numbers of s_j's buddies. EVEREST's division of disk space guarantees the existence of $B - 1$ buddy sections physically adjacent to s_j.
4. For every buddy $s_k, k \leq 0 \leq B - 1, k \neq j$, if it exists on the free list then mark it.
5. Any of the s_k unmarked buddies currently store parts of other object(s). The space must be re-arranged by swapping these s_k sections with those on the free list. Note that for every buddy section that should be freed there exists a section on the free list. After swapping space between every unmarked buddy section and a free list section, enough contiguous space has been acquired to create a section at height $i + 1$ of the free list.
6. Go back to Step 1.

To illustrate, consider the organization of space in Figure 4a. The initial set of disk resident objects is $\{o_1, o_2, o_3\}$ and the system is configured with $B = 2$. In Figure 4a, two sections are on the free list at height 0 and 1 (addresses 7 and 14 respectively), and o_3 is the victim object that is deleted. Once block 13 is placed on the free list in Figure 4b, the number of sections at height 0 is increased to B and it must be compacted according to Step 1. As sections 7 and 13 are not contiguous, section 13 is elected to be swapped with section 7's buddy, i.e., section 6 (Figure 4c). In Figure 4d, the data of section 6 is moved to section 13 and section 6 is now on the free list. The compaction of sections 6 and 7 results in a new section with address 6 at height 1 of the free list. Once again, a list of length two at height 1 violates Property 3.1 and blocks (4,5) are identified as the buddy of section 6 in Figure 4e. After moving the data in Figure 4f from blocks (4,5) to (14,15), another compaction is performed with the final state of the disk space emerging as in Figure 4g.

Once all sections of a deallocated object are on the free list, the iterative algorithm above is run on each list, from the lowest to the highest height. The previous algorithm is somewhat simplified because it does not support the following scenario: a section at height i is not on the free list, however, it has been broken down to a lower height (say $i - 1$) and not all subsections have been used. One of them is still on the free list at height $i-1$. In these cases, the free list for height $i-1$ should be updated with care because those free sections have moved to new locations. In addition, note that the algorithm described above actually performs more work than is strictly necessary. A single section of a small height, for example, may end up being read and written several times as its section is combined into larger and larger sections. This can be eliminated in the following manner. The algorithm is first performed "virtually" — that is, in main memory, as a compaction algorithm on the free lists. Once completed, the entire sequence of operations that have been performed determines the ultimate destination of each of the modified sections. These sections are then read and written directly to their final locations. One may observe that the total amount of data that is moved (read and then written) during any compaction operation is no more than $B-1$ times the total amount of free space on the list. For example, when $B=2$ then in the worst case, the number of bytes written due to preventive operations is no more than the number of bytes materialized, in an amortized sense. One may expect, however, that for a collection of objects of varying sizes, this number to be smaller.

The value of B impacts the frequency of preventive operations. If B is set to its minimum value (i.e., $B=2$), then preventive operations would be invoked frequently because every time a new section is enqueued there is a 50% chance for a height of the free list to consist of two sections (violates Property 3.1). Increasing the value of B will therefore "relax" the system because it reduces the probability that an insertion to the free list would violate Property 3.1. However, this would increase: 1) the number of seeks observed when retrieving an object, and 2) the expected number of bytes migrated per preventive operation. For example, at the extreme value of $B=n$ (where n is the total number of blocks), the organization of blocks will consist of two levels, and for all practical purpose, EVEREST reduces to a variant of Standard.

The design of EVEREST suffers from the following two limitations. First, it incurs a fixed number of seeks (although few) when reading an object. Second, the overhead of its preventive operations may become significant if many objects are swapped in and out of the disk drive (this happens when the working set of an application cannot become resident on the disk drive). The primary advantage of the elaborate object deallocation technique of EVEREST is that it avoid internal and external fragmentation of space as described for traditional buddy systems (see [GR93b]).

3.3.4 Implementation Considerations

In an actual implementation of EVEREST, it might be infeasible to fix the number of blocks as an exact power of B. Rather, one would generally fix the block size of the file system in a manner dependent upon physical characteristics of both the device and the objects in the database. This is possible with some minor modifications to EVEREST. The most important implication of an arbitrary number of blocks is that some sections may not have the correct number of buddies ($B-1$ of them). However, we can always move those sections to one end of the medium — for example, to the side with the highest block-offsets. Then instead of choosing the first section in Step 2 in the object deallocation algorithm (Section 3.3.3), one should choose the one with the lowest block-number. This ensures that the sections towards the critical end of the disk — that might not have the correct number of buddies — are never used in both Steps 4 and 5 of the algorithm.

3.4 REBATE

REBATE [GI94] partitions the available space of a device i into g regions ($\mathcal{G}_1$, $\mathcal{G}_2$, ..., $\mathcal{G}_g$) by analyzing: 1) the storage capacity of the device, termed C_i, and 2) the size and frequency of access to each object in the database, termed size(o_x) and heat(o_x) respectively [CABK88]. Each region $\mathcal{G}_j$ occupies a contiguous amount of space. The amount of space allocated to region $\mathcal{G}_j$ (termed space($\mathcal{G}_j$)) is determined such that the overall utilization of the space is maximized, i.e., the probability of a byte from a region containing useful data — data which is most likely to be accessed in the future — is maximized and is approximately the same for all regions. Each region manages a set of unique objects (OBJ($\mathcal{G}_j$) $= \{o_1, o_2, \ldots, o_k\}$). The minimum and maximum size of an object managed by a region $\mathcal{G}_j$ (termed min($\mathcal{G}_j$) and max($\mathcal{G}_j$) respectively) are unique. The space allocated to region $\mathcal{G}_j$ is split into ℓ_j fixed sized frames, where $\ell_j = \lfloor \frac{\text{space}(\mathcal{G}_j)}{max(\mathcal{G}_j)} \rfloor$. All objects whose size lie in the range from min($\mathcal{G}_j$) to max($\mathcal{G}_j$) are managed by region $\mathcal{G}_j$ and compete for its frames. REBATE wastes disk space when the size of a frame is larger than its occupying object. In order to minimize this waste, the regions are constructed so that the size of all objects in OBJ($\mathcal{G}_j$) is approximately the same. (Hence REBATE attempts to minimize the value $max(\mathcal{G}_j) - min(\mathcal{G}_j)$ for each region $\mathcal{G}_j$.) If o_x maps to region $\mathcal{G}_j$ and does not currently occupy a frame of $\mathcal{G}_j$ then the system compares work(o_x) with the other objects that currently occupy a frame of $\mathcal{G}_j$. It replaces the object with the least imposed work (say object o_y) only if work(o_y) < work(o_x). Otherwise, o_x does not become resident on this stratum. Further details are presented in [GI94], which also provides an efficient dynamic programming algorithm for constructing optimal *region-based partitions* when accurate data on the heat of objects is available.

With REBATE, the system might be required to either construct new regions or re-allocate space among the existing regions for at least two reasons. First, a new object might be introduced whose size is larger than the size of objects that constitute the present database. In this case, none of the existing frames can accommodate this object, and so a new region of larger frames must be introduced. Second, the access pattern to the objects might evolve in a manner that dictates the following: one or several of the current regions deserves more space than already allocated to it, while other regions deserve proportionally less space. Hence the design of REBATE includes a re-organization technique that periodically proposes a new organization of the regions, and renders it effective only if its expected improvement in the actual hit ratio observed by the device (that is, the effective utilization of its space) exceeds a preset threshold. This on-line re-organization procedure is described further in [GI94].

REBATE may suffer from two limitations. First, in a system where the objects sizes are not naturally clustered into like-sized classes, REBATE may waste space. This under-utilization of available space in turn increase the frequency of access to the tertiary storage device (when compared to Standard that packs objects on the disk drive without ensuring their contiguity). Yet even in the case where such natural classes exists, determining a truly optimal partition of the device's space amongst regions is an NP-hard problem. REBATE's compromise — settling for region-based partitions — may in fact be suboptimal in certain worst-case scenarios [GI94]. Overall, whether REBATE outperforms Standard depends on a number of factors including: the amount of wasted space; the size of the working set of an application relative to the capacity of the device; the overhead attributed to performing seeks when retrieving an object; and the penalty incurred in accessing the device at the next stratum. (With our assumptions, the impact of last factor is significant.)

Second, REBATE partitions the available disk space among multiple regions, necessitating a re-organization process when deployed for a database where the frequency of access to its objects varies dynamically over time. This re-organization procedure is undesirable for several reasons. First, the overhead of re-organization can be expensive if it evaluates alternative layout of regions (by invoking the REBATE algorithm) too frequently. Second, the re-organization procedure can only respond after it has detected a lower hit ratio than is expected. Consequently, the user must observe a *higher* latency than expected for a while before the re-organization procedure can recognize and remedy the situation. Third, the re-organization procedure will almost certainly fail in environments where the frequency of access to the objects changes in a manner that forces a frequent reallocation of space among regions (a "ping–pong" effect, in which space is shuffled back and forth to follow the regions of hottest objects). When these frequencies change too often, an even worse situation arises where the re-organization procedure instantiates a new layout corresponding to heat values that have already changed. The system is thus trying to "predict the future"; yet its only guide in this task is the statistical information that it can accumulate. When these quantities are unreliable, or show large and frequent variation, these predictive methods are bound to fail. In this circumstance, the use of an on-line re-organization method can itself cause further degradation in the system's performance.

4 Performance Evaluation

To quantify the performance tradeoffs of Standard, Dynamic, EVEREST and REBATE, we conducted a number of simulation studies. The simulation model evolved over a period of twelve months. During this period, we conducted many experiments and gained insights into: 1) the factors that impact the performance of the system with alternative space management techniques, 2) the experimental design of the simulator, and 3) the results that were important to present. Indeed, the design of EVEREST was introduced once we had understood the tradeoffs associated with Standard, Dynamic, and REBATE.

Almost all the components of the simulator are straightforward, except for the Driver module that generates requests, with each request referencing an object. It is complicated because it employs several distributions to generate the requests pertaining to an arbitrary pattern of access to the objects. An arbitrary request generator was desirable for several reasons. First, it eliminated the possibility of bias towards a technique. Second, we believe that it models reality because the pattern of access to the objects is typically unknown in real applications. Third, using this model, it is straightforward to evaluate the accuracy of the statistical modules for estimating heats. We observed that the statistics module is fairly accurate.

The design of the Driver is based on the assumption that the heat of objects evolves gradually. For example, one may sample the distribution of access to the objects at two different points in time and observe that 80% of requests are directed to 20% of the objects for the first sample (termed 80-20 access pattern) and a 90-10 access pattern for the second sample (90% of accesses are directed to 10% of the objects). Our assumption states that the heat evolved incrementally and that at some point it was more uniform than both 80-20 and 90-10 access patterns. The Driver models this paradigm by using a normal distribution to model each of 80-20 and 90-10 access patterns. Next, it migrates the heat of objects in 10 steps from 80-20 to 90-10. After the first interval, the distribution is more uniform than both 80-20 and 90-10. It is most uniform at the fifth interval. Starting with the sixth interval, the Driver starts its progress towards a 90-10 access pattern. By the tenth interval, the Driver is producing requests to the objects based on a 90-10 access pattern. (The details of this is provides in Section 4.1.) We investigated simpler designs for generating requests (e.g., changing the distribution of access from 80-20 to 90-10 in one step) and observed no change in the final conclusions.

Early on, we employed the average service time observed with alternative space management techniques as the criterion to compare one strategy with another. This was a mistake because it hid the factors that impact the performance of the system with alternative techniques by associating weights to them. (These weights describe the physical characteristics of the devices in the hierarchy.) By focusing on these factors (instead of the average service time), we were able to develop analytical models that incorporate the physical parameters of a system to compute the average service time (see Section 5). These analytical models were validated using the simulation study (with less than 2% margin of error). Using these models, the system designer may choose the value of parameters corresponding to a target hardware platform to evaluate alternative techniques.

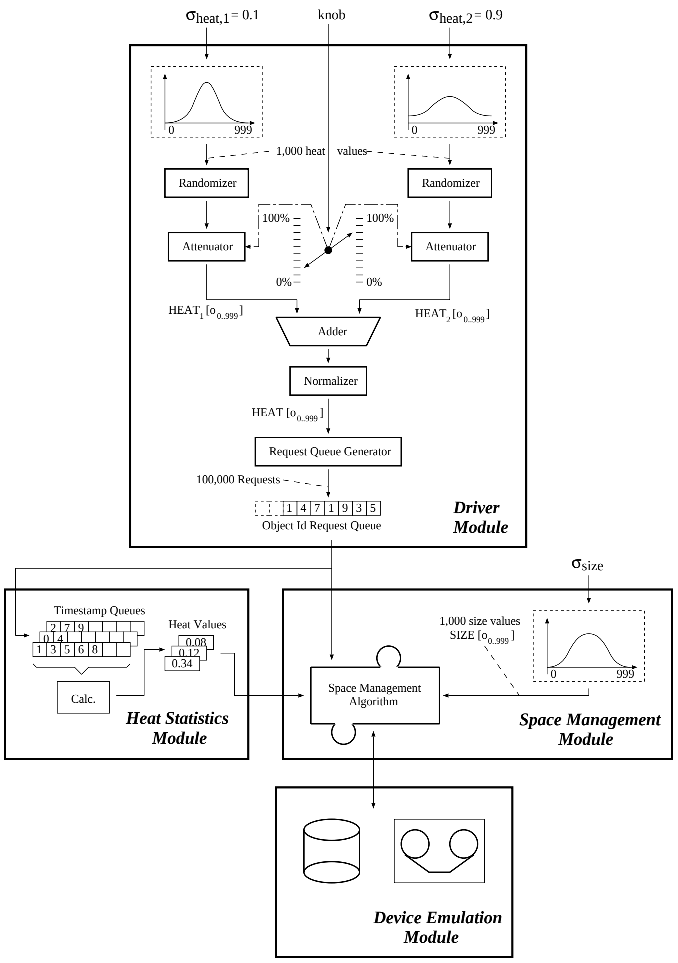

Figure 5: Block diagram of the Simulation Model.

4.1 Simulation Model

The simulation model consists of four components: Driver, Space Management, Device Emulation, and Heat Statistics (see Figure 5). The *Driver* module generates a synthetic workload by constructing a queue of object requests. The *Space Management* module realizes the different space management algorithms and interfaces with the *Device Emulation* module. The Device Emulator models a magnetic disk drive with its seeks and transfer times (see [RW94]) and a simple tertiary device (i.e., a tape drive). Finally, the *Heat Statistics* module gathers information about the sequence of requests and compiles this data into a heat value for every object. The estimated heat value is then used by the Space Management module to decide which objects should be disk resident. The simulator was implemented using the C programming language.

The Driver module uses three input parameters to generate a sequence of object requests: two heat distributions ($\sigma_{heat,1}$ and $\sigma_{heat,2}$), and a *knob*. The knob determines the role of a given σ_{heat} in generating the requests. As illustrated in Figure 5, when knob is equal to 0% for $\sigma_{heat,2}$, its value is 100% for $\sigma_{heat,1}$ (in this case, the value chosen by $\sigma_{heat,1}$ determines the final queue of requests). To describe how requests are generated using a normal distribution, assume that knob is equal to 0% for $\sigma_{heat,2}$. In this case, object heats (or frequency of their appearance in the request queue) obey a normal distribution with a mean of zero and a standard deviation of $\sigma_{heat,1}$. Objects are viewed as uniformly distributed sample points in the interval [-1,1]. With a small value of $\sigma_{heat,1}$, the access pattern is skewed, and most accesses will be concentrated on a smaller subset of the objects. With larger values of σ_{heat}, the heats of the objects becomes more uniform. As a rule of thumb, approximately 60% of the heat will be concentrated on a $\sigma_{heat,1}$ fraction of all objects, and 90% of the heat on a $2\sigma_{heat,1}$ fraction. For example, when $\sigma_{heat,1}$=0.1, then approximately 98% of the heat is concentrated in 25% of the objects. As the value of $\sigma_{heat,1}$ increases, this ratio changes rapidly: At $\sigma_{heat,1}$=0.2, nearly 78% of the heat is concentrated in 25% of the objects, while at $\sigma_{heat,1}$=0.4, less than 50% of the heat is concentrated on 25% of the objects. For values of $\sigma_{heat,1} \geq 1$, this distribution is nearly uniform. Objects are assigned heats in a purely random manner; there is no intentional correlations between the size and heat of objects.

The Driver uses two heat distributions to generate the final queue of requests. Both distributions are based on the same normal distribution with a mean of zero. The knob controls to what extent each of the two heat distributions is used in generating the request queue. When the value of knob changes, the heat essentially migrates from one set of objects to another. At one end, 0% of the $\sigma_{heat,1}$-curve and 100% of the $\sigma_{heat,2}$-curve are in effect. At the other end, the percentages are reversed, i.e., 100% of the $\sigma_{heat,1}$-curve and 0% of the $\sigma_{heat,2}$-curve are used. For every object o_x, heat$_1(o_x)$ and heat$_2(o_x)$ are added and stored in the array heat$(o_{0...999})$. This array of results is further normalized such that $\sum_{j=0}^{999} \text{heat}(o_j) = 1$. Finally, the request queue is generated from the heat$(o_{0...999})$ array.

The Space Management module services the requests that are generated by the Driver module. It has access to a synthetic database that consists of 1,000 objects. A normal distribution of the sizes guarantees a fixed average object size for all experiments (controlled by the input parameter σ_{size}).

Each different space management algorithm that implements Standard, Dynamic, EVEREST, and REBATE policies is a plug-in module.

The Device Emulation part of the simulator consists of data structures and routines to emulate a magnetic disk drive and a tertiary device. We employed the analytical models of [RW94] to represent the seek operation, the transfer rate and latency of a magnetic disk drive. The tertiary device is simplified and only its transfer rate is modeled. This module is also responsible for gathering the statistics that are used to compare the effectiveness of the different space management policies (the number of seeks performed on behalf of an object, the average percentage of disk space that remains idle).

The Space Management module does not have access to any heat information that exist in the Driver and must learn about it by gathering statistics from the issued requests. The learning process is as follows. The module keeps a queue of timestamps for every object as well as an estimated heat value. All the queues are initially empty and the heat values are uniformly set to $\frac{1}{n}$, where n is the total number of objects. Upon the arrival of a request referencing object o_x, the current time is recorded in the queue of object o_x. Whenever the timestamp queue of object o_x becomes full, the heat value of that object is updated according to

$$\text{heat}_{new}(o_x) = (1-c) \times \frac{1}{\frac{1}{K} \times \sum_{i=1}^{K-1} t_{i+1} - t_i} + c \times \text{heat}_{old}(o_x)$$

where K is the length of the timestamp queue (set to 50), c is a constant between 0 and 1 (set to 0.5), and t_x is one individual timestamp. After the update is completed, the queue of this object is flushed and new timestamps can be recorded. This approach is similar to the concept of the *Backward K-distance* used by the authors of [OOW93] in the LRU-K algorithm. The two schemes differ in three ways. First, the heat estimates are not based on the interval between the first and the last timestamp in the queue but are averages over all the intervals. Second, the heat value of an object o_x is only updated when the timestamp queue of o_x is full, therefore reducing overhead. And third, the previous heat value $\text{heat}_{old}(o_x)$ is by a fraction of c taken into account when $\text{heat}_{new}(o_x)$ is calculated. The above measures balance the need for smoothing out short term fluctuations in the access pattern and guaranteeing responsiveness to longer term trends.

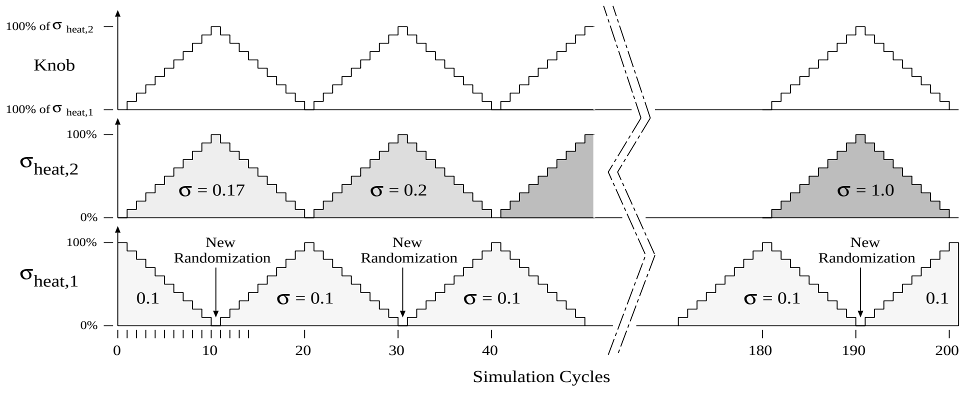

Figure 6: Values over time of three of the input parameters for the simulation experiments.

4.2 Experimental Design

The two simulation model input parameters $\sigma_{heat,1}$ and $\sigma_{heat,2}$ are used to model how the heat of individual objects might change over time. The relevant parameters of the experiments are summarized in Table 2. The value of $\sigma_{heat,1}$ is always held constant at 0.1. The parameter $\sigma_{heat,2}$ is initially set to 0.17. The value of the knob is initialized to 100% of $\sigma_{heat,1}$. After 100,000 requests the value of knob is decremented by 10% (and therefore the ratio of requests corresponding to $\sigma_{heat,2}$ increased from 0% to 10% and that of $\sigma_{heat,1}$ to 90%). This process continues with the knob value decreasing by 10% after every 100,000 requests until its value reaches 0% (at this point, all requests correspond to the $\sigma_{heat,2}$ distribution). The heat represented by $\sigma_{heat,1} = 0.1$ is now re-distributed by invoking the randomization routine. At this point the value of knob starts to increase by 10% increments. Each time a new queue of requests is generated. This procedure is repeated many times. At extreme values of knob for $\sigma_{heat,1}$ (i.e., 0% and 100% for $\sigma_{heat,1}$), a random number generator is employed to ensure that the identity of frequently accessed objects changes, requiring the system to learn the identity of the frequently accessed objects each time.

The above experiment was repeated a total of 10 times, each time with a different $\sigma_{heat,2}$ parameter. The values used are listed in Table 2 and Figure 6 illustrates the process.

Device Parameters	
Disk Size	1 GB
Database (also Tertiary) Size	4 GB
Block Size (where applicable)	4 kB
Object Parameters	
Number of Objects	1,000
Maximum Object Size	7.9 MB
Minimum Object Size	0.1 MB
Average Object Size	4.0 MB
Input Parameters	
σ_{size}	0.3
$\sigma_{heat,1}$	0.1
$\sigma_{heat,2}$	0.17, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0

Table 2: Simulation Parameters.

4.3 Performance Results

Figure 7 presents the number of seeks observed per request that finds its referenced object on the disk drive[5] (termed a disk hit). With an empty disk, Standard lays the referenced object contiguously. However, after a few iteration of knob changing its value, each request observes on the average more than 60 seeks. Dynamic and REBATE ensure a contiguous layout and observe zero seeks per disk hit. As expected, due to its preventive style, EVEREST renders the number of seeks a constant (4.5 in this experiment; this number represents the total seeks required to both service a request observing a disk hit and the preventive operations performed by EVEREST).

Figure 8 demonstrates the disadvantages of laying out an object contiguously with Dynamic and REBATE. Dynamic wastes 1% to 3% of the available disk space (this is explained in Section 3.2). REBATE wastes approximately 14% of the disk space due to internal fragmentation of a frame. Both Standard and EVEREST utilize the available space to its fullest potential. They do waste a small fraction of space (less that 0.1%) due to our assumption that an object should be resident in its entirety (no partial materialization of an object is allowed).

Figure 9 quantifies the overhead attributed to the preventive characteristics of EVEREST. The number of preventive operations performed depends on how frequently the replacement policy is activated to locate and delete victim objects. To illustrate, the peaks in Figure 9a correspond to

[5]This number does not include the first seek required to access the first block of an object

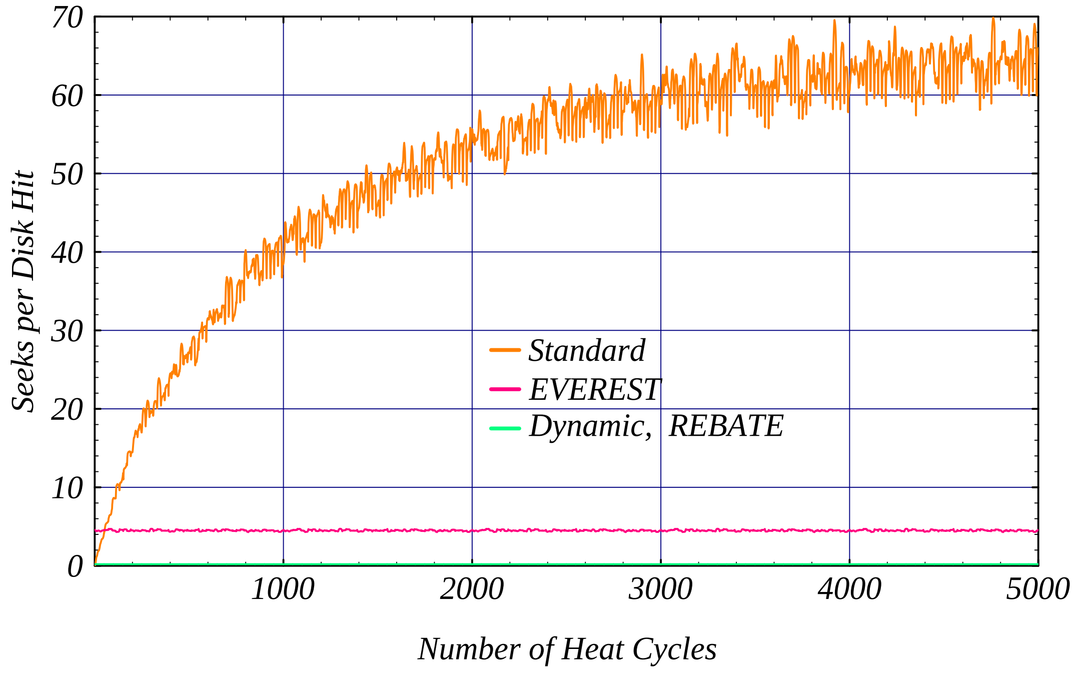

Figure 7: Number of seeks per disk hit.

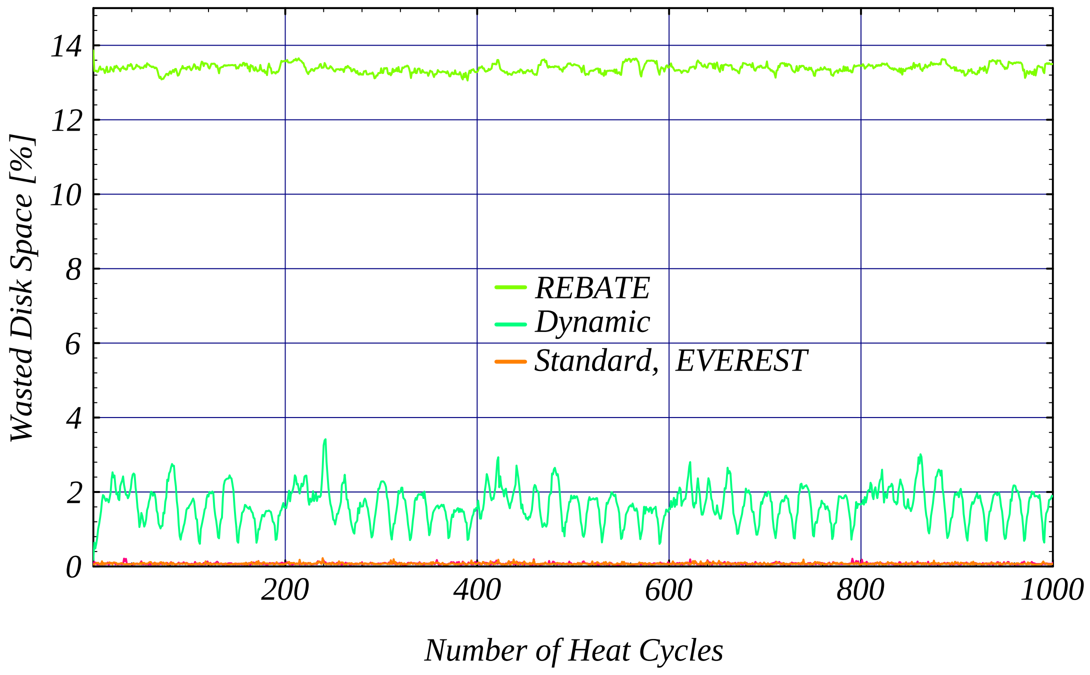

Figure 8: Wasted disk space.

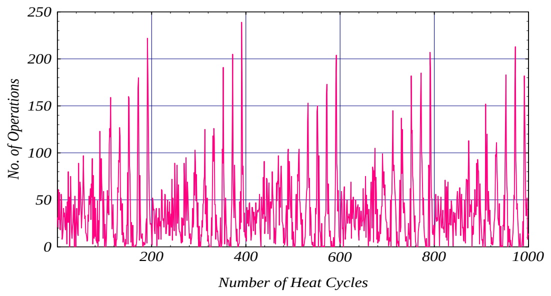

Figure 9a: The number of preventive operations.

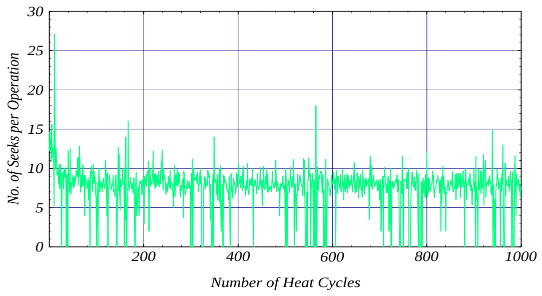

Figure 9b: The number of seeks per preventive operation. Each migration of a section requires two seeks (1 read + 1 write).

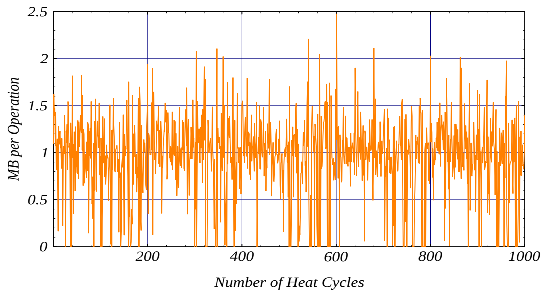

Figure 9c: MBytes of data migrated per preventive operation.

Figure 9: Overhead attributed to the preventive characteristic of EVEREST.

the value of $\sigma_{heat,2}$ (knob = 100% for $\sigma_{heat,2}$). As described in Section 4.2 (Table 2), the value of $\sigma_{heat,2}$ increases from 0.17 to 1. At $\sigma_{heat,2}$=1, the distribution of access to the objects is fairly uniform, motivating the replacement policy to delete several objects from the disk in favor of the others. The amount of work (disk activity) performed per preventive operation depends on the degree of fragmentation of sections on the disk drive. Figure 9b and 9c demonstrate the number of seeks incurred and the amount of migrated data attributed to a preventive operation. While there is significant variation, on the average, a preventive operation requires 8 seeks and the migration of 1 MByte of data (note, this is 25% of the average requested object size). Once amortized across all the requests, this overhead becomes negligible (as illustrated by the number of incurred seeks in Figure 7).

Figure 9 demonstrates the following. First, the number of preventive operations should be a small fraction of the total number of requests serviced by a device. This clearly states the need for the existence of a working set. Otherwise, the number of objects replaced may become significant and, in turn, cause the overhead attributed to the preventive nature of EVEREST to dominate the average service time of the device. Second, the latency incurred by a request might be variable depending on: 1) whether a preventive operation is invoked, and 2) the amount of work performed by this preventive operation.

Finally, we compared the obtained results with the scenario where the system was allowed access to the queue of requests and could compute the heat of the objects with 100% accuracy (as compared to employing the Heat Statistics Module to learn the heat information, see Section 4.1 for the details of this module). The obtained results were almost identical demonstrating that: 1) the technique employed by the Heat Statistics Module has no impact on the obtained results, and 2) the employed technique to compute the heat statistics is effective in our experimental design.

5 Analytical Models

In this section, we develop analytical models that approximate the average service time of a system based on (1) its physical characteristics and (2) the fundamental factors that impact the performance of the system with the alternative space management techniques. These abstract models are useful because a system designer may manipulate the value of their parameters to understand the benefits of one strategy as compared to another. They have been validated using the experimental simulation model.

The performance of the system with a space management strategy is impacted by the following factors:

1. Average number of seeks incurred when reading an object (F) and the average time to perform a seek (S_{Seek}).
2. Number of preventive operations performed (P) and the average time to perform one such operation (S_{Prev}).
3. Number of bytes re-organized (U) and the number of seeks attributed to the re-organization procedure (E).
4. The amount of wasted space (W) and its expected hit ratio.

We analyze the average service time of the system with a given strategy to service a fixed number

Term	Definition
C	Storage capacity of the magnetic disk
R_T	Total number of requests issued during a fixed period of time
R_H	Total number of requests that observe a disk hit during the fixed period of time
Size_{Avg_req}	Average number of bytes retrieved per request
B	Total number of bytes retrieved by R_T requests
H	Total number of bytes found on the disk by R_T requests
P	Average number of preventive operations per disk hit
W	Total number of bytes wasted by a strategy
U	Total number of bytes re-organized (read + write)
E	Total number of seeks attributed to the re-organization procedure
F	Average number of seeks per disk hit
$D_{Tertiary}$	Delivery rate of tertiary storage device (incorporates the seek time of the device)
T_{Disk}	Transfer rate of the disk drive
byte_hit	Fraction of a byte that observed a hit, byte_hit $= \frac{H}{B}$
S_{Seek}	Average service time for a seek
S_{Prev}	Average service time for a preventive operation

Table 3: List of parameters used by the analytical models.

of requests and quantify what fraction of this service time is attributed to each of these factors. One or more of these factors might be non-existence for a strategy. For example, REBATE incurs the overhead of neither preventive operations nor the seeks attributed to retrieval of an object. In this case, the value of appropriate parameters will be zero (F and P for REBATE) enabling the model to eliminate the impact of these factors. (Refer to Table 1 for a list of factors attributed to the different space management techniques described in this paper.) We assume that the system has accumulated the statistics shown in Table 3. We describe each factor and its corresponding analytical model in turn.

The portion of average service time attributed to seeks incurred when reading an object is:

$$F \times S_{Seek} \tag{2}$$

where F is the average number of seeks performed on behalf of a retrieval from disk, and S_{Seek} is the average service time to perform a seek. These statistics can be gathered as the system services requests.

The portion of average service time attributed to preventive operations is defined as:

$$P \times S_{Prev} \tag{3}$$

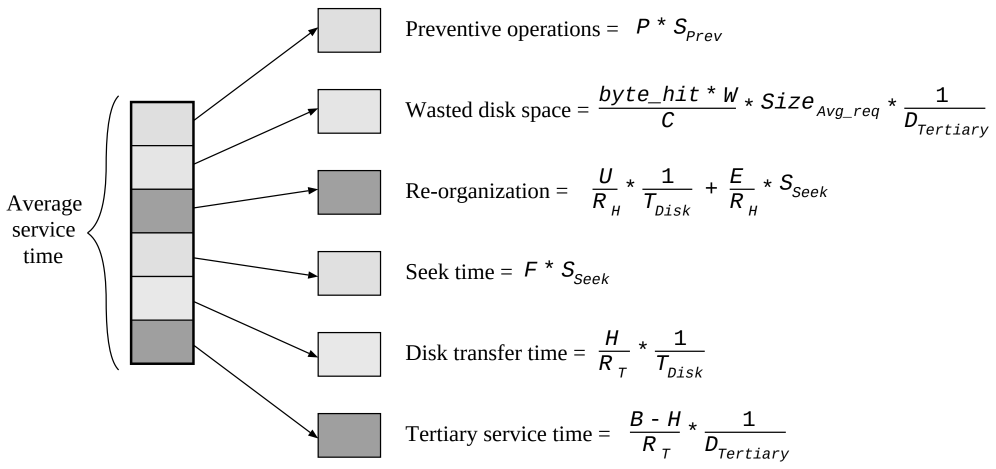
Average service time
Preventive operations = $P * S_{Prev}$
Wasted disk space = $\frac{byte\_hit * W}{C} * Size_{Avg\_req} * \frac{1}{D_{Tertiary}}$
Re-organization = $\frac{U}{R_H} * \frac{1}{T_{Disk}} + \frac{E}{R_H} * S_{Seek}$
Seek time = $F * S_{Seek}$
Disk transfer time = $\frac{H}{R_T} * \frac{1}{T_{Disk}}$
Tertiary service time = $\frac{B - H}{R_T} * \frac{1}{D_{Tertiary}}$

Figure 10: Components of average service time for a single queue of requests.

P defines the average number of preventive operations per disk hit, and S_{Prev} is the average service time to perform a preventive operation.

The amount of time attributed to disk transfer time is a function of the average number of bytes retrieved from the disk per request ($\frac{H}{R_T}$) and the transfer rate of the disk drive (T_{Disk}):

$$\frac{H}{R_T} \times \frac{1}{T_{Disk}} \tag{4}$$

Similarly, the amount of time spent transfering data from tertiary is a function of the average number of bytes retrieved from the tertiary per request ($\frac{B-H}{R_T}$) and the delivery rate of the tertiary storage device:

$$\frac{B - H}{R_T} \times \frac{1}{D_{Tertiary}} \tag{5}$$

The delivery rate of tertiary storage device incorporates the average number of seeks incurred by this device per request, and the overhead of such seeks.

A technique that employs a re-organization process, reads and writes a fixed number of bytes (U) causing the device to incur a fixed number of seek operations (E). This overhead averaged across all requests (R_H) is:

$$\frac{U}{R_H} \times \frac{1}{T_{Disk}} + \frac{E}{R_H} \times S_{Seek} \tag{6}$$

A technique such as REBATE may waste disk space. However, its impact might be negligible if the wasted space is not expected to have a high hit ratio. Assume the existence of a unit that defines what fraction of each byte on the disk should observe a hit, termed byte_hit ratio (its details are presented in the following paragraphs). The wasted space reduces the change in byte_hit by a fixed margin: $\frac{byte\_hit \times W}{C}$. This causes a fixed number of bytes of the average request to be retrieved from the tertiary storage device ($\frac{byte\_hit \times W}{C} \times Size_{Avg\_req}$), and the overhead of reading these bytes can be quantified as:

$$\frac{byte\_hit \times W}{C} \times Size_{Avg\_req} \times \frac{1}{D_{Tertiary}} \tag{7}$$

Byte hit ratio is a function of the size of both the working set of the system and the storage capacity of the disk drive. When the size of the working set of an application is larger than the storage capacity of the disk, every byte becomes valuable because it minimizes the number of bytes retrieved from the tertiary storage device. In this case, byte hit ratio is defined as: $\frac{H}{B}$. When the working set is smaller than the storage capacity of the disk, the probability of a wasted byte observing a hit is a function of the database size, the amount of wasted space, and the pattern of access to the objects. For example, if one assumes that references are randomly distributed across the objects (bytes) that constitute the remainder of database (except for those that are part of the working set), then byte_hit might be defined as: $\frac{1}{size(DB)-W}$.

We verified the analytical models using the simulator. This was achieved as follows. The simulation was invoked for a period of time in order to accumulate the value of parameters outlined in Table 3. Next, the average service time of the system as computed by the simulator was compared with that of the analytical model. In almost all cases, there was a perfect match. The highest observed margin of error was less than 2%. It is important to note that these models should be extended with queuing times in the presence of both multiple users and multiple disk drives (this is beyond the focus of this study).

6 Conclusions

In this paper we have studied alternatives in space management for large repositories of objects that are generally retrieved sequentially and in their entirety. These repositories might be found in various multimedia applications (such as video–on–demand servers) and in numerous scientific applications (such as the Brookhaven and Cambridge database of molecular structures). To isolate those factors that contribute significantly to the performance of such a system, we have sampled the space of storage management policies for a fixed hierarchical architecture. The simulation results, and their analyses, permit one to isolate the trade-offs inherent in various designs: trade-offs between wasted time (seeking) and wasted space; between local, greedy techniques for optimizing a device's workload (as in Standard or Dynamic) and those that impose a more global order on the medium (REBATE or EVEREST); between detective and preventive strategies for adapting to a changing workload. However, a complete evaluation of these trade-offs is dependent on both the physical characteristics of the system and the target application. For example, the impact of wasted space and wasted time upon the actual workload of the device depends critically on its seek time and bandwidth, block size and average object size, and the size of the resident working set relative to the capacity of the device. Whether a system should impose a global order on its device, and effectively partition its space, may depend upon the changeability of the working set and the expected or observed variance in the heats of objects. Similarly, the policy adopted to organize and re-organize space depends upon characteristics of the tasks for which it is deployed: whether a working set exists, how quickly it changes, and how predictable its evolution is.

While we believe that this study is complete in its treatment of issues that arise with design of strategies to manage the space of a mechanical device, it raises two related research topics that deserve further investigation. First, implementation details of Dynamic, REBATE and EVEREST are lacking and require further consideration should a system designer elect to employ one of these strategies in a file system. In particular, we intend to investigate the crash-recovery component of these strategies (i.e., it enables the device to recover to a consistent state after a power failure). Second, the management of objects across the different strata of a hierarchical storage structure requires further analysis. In particular, a management technique should decide if it is worthwhile to allow multiple copies of an object with one copy residing at a different stratum of the hierarchy (e.g., one copy on the magnetic disk and a second on the optical disk in Figure 1).

References

[BGMJ94] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju. Staggered striping in multimedia information systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[BKW+77] F.C. Bernstein, T.F. Koetzle, G.B. Williams, E.F. Mayer, M.D. Bryce, J.R. Rodgers, O. Kennard, T. Himanuchi, and M. Tasumi. The Protein Databank: A Computer Based Archival File for Macromolecular Structures. *Journal of Mol. Biol.*, 112(2):535–542, 1977.

[CABK88] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 100–110, 1988.

[CDKK85] H. T. Chou, D.J. DeWitt, R. Katz, and T. Klug. Design and implementation of the Wisconsin Storage System. *Software Practices and Experience*, 15(10), 1985.

[CHL93] M. Carey, L. Haas, and M. Livny. Tapes hold data, too: Challenges of tuples on tertiary storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–417, 1993.

[CL93] H.J. Chen and T. Little. Physical Storage Organizations for Time-Dependent Multimedia Data. In *Proceedings of the Foundations of Data Organization and Algorithms (FODO) Conference*, October 1993.

[Cou88] National Research Council. Mapping and Sequencing the Human Genome. In *Committee on the Human Genome Board on Basic Biology*. National Academy Press, April 1988.

[Den68] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[Gal91] D. Le Gall. MPEG: a video compression standard for multimedia applications. *Communications of the ACM*, April 1991.

[GI94] S. Ghandeharizadeh and D. Ierardi. Management of Disk Space with REBATE. *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM)*, November 1994.

[Gib92] G. A. Gibson. *Redundant Disk Arrays: Reliabls Secondary Storage.* MIT Press, Cambridge, Mass., 1992.

[GR93a] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, chapter 13, pages 670–671. Morgan Kaufmann, 1993.

[GR93b] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, chapter 13, pages 682–684. Morgan Kaufmann, 1993.

[Kno65] K. C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, October 1965.

[LD91] H. R. Lewis and L. Denenberg. *Data Structures & Their Algorithms*, chapter 10, pages 367–372. Harper Collins, 1991.

[MJLF84] M. Mckusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, August 1984.

[NY94] R.T. Ng and J. Yang. Maximizing Buffer and Disk Utilizations for News On-Demand. In *Proceedings of the International Conference on Very Large Databases*, September 1994.

[OOW93] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–417, 1993.

[PGK88] D. Patterson, G. Gibson, and R. Katz. A case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1988.

[RO92] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *Transactions on Computer Systems*, 10(1):26–52, 1992.

[RW94] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, March 1994.

[TPBG93] F.A. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming RAID-A Disk Array Management System for Video Files. In *First ACM Conference on Multimedia*, August 1993.

A Display of Continuous Media

This appendix explains the significance of predicting the service time of a mechanical device (e.g., a disk drive) in order to schedule it effectively to display an object of continuous media data type (e.g., video). To simplify the discussion, assume that the target environment consists of some memory, one disk drive and a single tertiary storage device (as described in Section 2). Moreover, assume that the bandwidth of both the tertiary storage device and magnetic disk drive exceed the bandwidth required to display (compressed) video objects. To ensure a continuous display of a video object X and minimize the amount of required memory, several studies [BGMJ94, NY94, CL93, TPBG93] have proposed that X be striped into n equi-sized subobjects (X_1, X_2, ..., X_n). Each subobject X_i represents a contiguous portion of X. To display X from a device (say the disk drive), the system stages X_1 from the disk drive to memory. It schedules the disk drive to read X_2 such that it becomes memory resident before the display of X_1 completes. Next, it initiates the display of X_1. This process is repeated for each X_i and X_{i+1} until all subobjects of X are displayed. For complete details of this mechanism see [BGMJ94].

In order to schedule the disk drive to satisfy the constraint that X_{i+1} is rendered memory resident before the display of X_i completes, the system must compute: 1) the display time of X_i, and 2) the time required to read X_{i+1} from the disk drive. Consider each factor in turn. The display time of X_i is a function of its size and the bandwidth required to display it. If the bandwidth required to display X is 1.5 mbps and the size of each of its subobjects is 100 KByte then the display time of X_i is 0.53 seconds. The size of subobjects of different media types is proportional to their bandwidth requirement [BGMJ94]. For example, if the bandwidth required to display object Y is 3 mbps then the size of each of its subobjects would be twice that of X (200 KBytes), however, note that their display time is identical simplifying the scheduling of time. Thus, in a database consisting of n media types each with a unique bandwidth requirement, one would find n classes of subobjects each with a unique size.

The service time of the disk drive to read subobject X_{i+1} is dependent on its size, the transfer rate of the disk and the number of disk seeks incurred when reading it. The number of seeks introduces variability in service time among the subobjects of X. These seeks can be eliminated by storing each subobject contiguously on the disk drive. In a file system based on Standard where the available disk space is partitioned into fixed size pages, the contiguously of a subobject is ensured when its size is smaller than that of a disk page. Otherwise, one may not assume that a subobject is stored contiguously. This is because the subobjects are swapped in and out of the available disk space depending on their expected future frequency of access in order to minimize the number of accesses to the tertiary storage device providing the user with a low latency time. However, this causes the disk space to become fragmented over a period of time. This forces the system to either store a subobject in a non-contiguous manner (introducing seeks) or delete more data (essentially subobjects that correspond to other objects) than necessary to ensure the contiguous layout of each new subobject. In the first case, the seeks are undesirable. In the second scenario, the number of references to the tertiary storage device increases as the deleted subobjects may have high expected future access. REBATE and Dynamic ensure a contiguous layout of each object. EVEREST bounds the number of seeks performed when retrieving an object in order to enable a scheduler to compute an upper bound on the service time of the disk.