Background data movement in a log-structured disk subsystem

by B. McNutt

The log-structured disk subsystem is a new concept for the use of disk storage whose future application has enormous potential. In such a subsystem, all writes are organized into a log, each entry of which is placed into the next available free storage. A directory indicates the physical location of each logical object (e.g., each file block or track image) as known to the processor originating the I/O request. For those objects that have been written more than once, the directory retains the location of the most recent copy. Other work with log-structured disk subsystems has shown that they are capable of high write throughputs. However, the fragmentation of free storage due to the scattered locations of data that become out of date can become a problem in sustained operation. To control fragmentation, it is necessary to perform ongoing *garbage collection*, in which the location of stored data is shifted to release unused storage for re-use. This paper introduces a mathematical model of garbage collection, and shows how collection load relates to the utilization of storage and the amount of locality present in the pattern of updates. A realistic statistical model of updates, based upon trace data analysis, is applied. In addition, alternative policies are examined for determining which data areas to collect. The key conclusion of our analysis is that in environments with the scattered update patterns typical of database I/O, the utilization of storage must be controlled in order to achieve the high write throughput of which the subsystem is capable. In addition, the presence of data locality makes it important to take the past history of data into account in determining the next area of storage to be garbage-collected.

1. Introduction

The log-structured disk subsystem is a new concept for the use of disk storage whose future application has enormous potential. In such a subsystem, all writes are organized into a log, each entry of which is placed into the next available free storage. A directory indicates the physical location of each logical object (e.g., each file block or track image) as known to the processor originating the I/O request. For those logical objects that have been written more than once, the directory retains the location of the most recent copy.

Over time, the older areas of the log become fragmented, as individual data objects stored there are rendered out of date. A de-fragmenting process (*garbage collection*) is needed to consolidate still-valid data and to release unused storage for re-use. Understanding the requirements of the garbage-collection process is among the interesting new challenges posed by log-structured disk technology. This paper investigates the amount of data movement which must be performed by the garbage-collection process, and how such data movement relates to the utilization of storage.

A simple, abstract model of the garbage-collection process is introduced that provides for some flexibility in the policy that identifies the specific data areas which should next be collected. Using this *generation group* model, we consider both the case in which the decision whether to collect a specific area of data is independent of the past history of the data, and the case in which this decision depends upon the number of times the data have previously been garbage-collected.

Rosenblum and Ousterhout [1] obtain a result, similar to (7) below, that relates garbage-collection load to the utilization of collected data areas, but not to the overall utilization of subsystem storage. The main contribution of the present paper is that, by introducing a realistic statistical description of the pattern of updates, we develop a method of analysis that ties garbage-collection load directly to overall storage utilization. In addition, this analysis approach is capable of examining both history-dependent and history-independent methods of garbage collection.

The mathematical modeling of the present paper complements and adds to the understanding of earlier simulation results obtained by the authors just mentioned [1, 2]. These include results addressing the behavior of garbage-collection load as well as the value of history-dependent garbage collection (although the exact form of history-dependent collection algorithm examined here is not identical).

We start, in Section 2, by briefly reviewing the concept of a log-structured subsystem and the role that garbage collection plays in the concept.

Section 3 then builds a framework for analysis by defining the generation group model. This section also explores the general properties of the model that apply regardless of the specific I/O workload. Unfortunately, the data movement required for garbage collection *does* depend upon the specific workload.

Section 4 continues by making the simplest possible workload assumption (without regard to realism), which we call the *linear model*. On the basis of the linear model, Section 4 develops a very simple formula for data movement due to garbage collection. This estimate shows that the load due to garbage collection depends critically upon the utilization of storage, and becomes unbounded as storage utilization approaches 100 percent. Only the history-independent method of segment selection is considered in Section 4.

Section 5 tests the result of Section 4 by using a more realistic workload model. This model, called *hierarchical reuse*, was introduced in an earlier paper as a way to describe cache reference locality. Using the hierarchical reuse model, Section 5 shows that

- The history-independent garbage-collection scheme imposes a somewhat higher level of garbage-collection activity than the linear model would predict, but
- By using the history-dependent strategy the cost can be brought down to a range of results of which most tend to be at or just below the linear prediction.

Thus, the linear model, despite its somewhat arbitrary assumptions, appears to be serviceable as a "rule-of-thumb" estimate of the garbage-collection load.

Finally, Section 6 concludes by considering the implications for using log-structured files to supplement a RAID-5 disk parity architecture. This strategy has recently been the focus of much attention because of its introduction into the high-end disk marketplace.

2. Overview of concepts

The concept of a log-structured disk subsystem (also referred to as a log-structured file system, or LSF) was first proposed by Ousterhout and Douglis [3] as a technique for improving the throughput of disk writes. The desire to improve write throughput was motivated by the observation that reads can often be serviced out of a cache memory, located either in the storage control or in the processor, without requiring a disk access; on the other hand, writes must eventually be copied to the disk storage medium to ensure permanence, even if cache memory is used to temporarily delay making the copy.

Rather than performing a write in the traditional manner, by locating the affected file block or track image and replacing its contents, Ousterhout and Douglis proposed that writes could be organized into a log. An entry of the log could simply be written into the next available storage, regardless of the previous location of the affected data object. A directory could then be used to keep track of the most recent copy of each data object.

An excellent practical overview of the architecture of a log-structured file system (SPRITE) and of the associated design issues is given by Rosenblum and Ousterhout [1]; extensive additional performance and other data for SPRITE are reported by Rosenblum [2]. To make possible the desired efficiency of writes, Rosenblum and Ousterhout group written data objects so that a reasonably large number can be transferred together to disk. In keeping with [1], we adopt the term *segment* for the storage needed

to write one such group of data objects. A segment, then, is the smallest usable area of contiguous free space; the log is written to disk storage one segment at a time.

Figure 1(a) diagrams the process of writing the log. As the figure shows, each log entry consists of a reasonably large number of data objects (usually more than the four objects used for purposes of illustration), and is copied as a unit into an empty segment of storage. The entries of the log need not be contiguous; instead, any empty segment can be selected to perform the next write (the exact choice of which empty segment is not important for our present purpose).

The data objects of a given log entry may render previously written data objects out of date. This is illustrated in the figure by using an empty circle to denote an out-of-date object; filled marks denote objects that are still current. At the moment in time presented by part (a) of the figure, one segment (the one most recently written) is filled completely with current data, two segments are half-filled, and two segments are empty. A log entry can be made only into one of the empty segments. In the example of the figure, the next entry (write D) will be placed into segment 4.

Figure 1(b) shows how write D might be used, not to store new data, but to perform garbage collection. This write operation copies four existing data objects, taken from the two half-full segments, into a single empty segment. Of the three segments affected, one becomes full and two are emptied.

As this example illustrates, garbage collection copies existing data in order to reduce the fragmentation of free space. The data copies made in performing garbage collection have the effect of adding to the overall number of log entries that must be made.

The modeling presented in this paper does not depend upon the specific type of data object going into a given segment, but we do assume that each segment contains a reasonably large *number* of data objects (large enough so that it makes sense to talk about this population in statistical terms). We also assume that there is no tendency for all of the objects in a segment to be updated at the same time, as might occur in an environment with mainly sequential files. Instead, the purpose of this paper is to examine the scattered update patterns typical of a database environment.

The metric that we adopt for the purpose of assessing the impact of garbage collection is the number of times a given data object must be moved during its lifetime. Since the life of each data object ends with a write operation that causes the object's log entry to be superseded, this metric is called called *moves per write*. Note that in evaluating the moves per write, all data object moves are counted equally. The actual efficiency with which each move can be performed is not examined in the present paper.

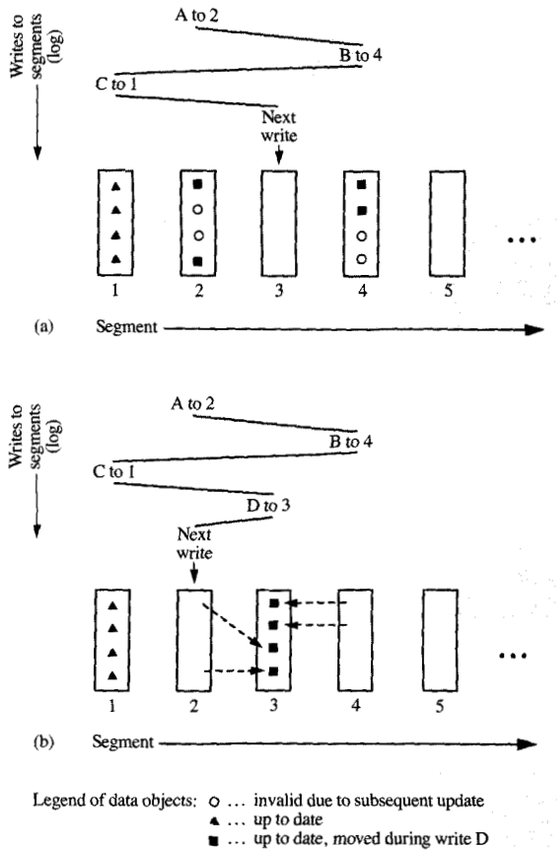

Figure 1

(a) Entries of the log are written into the next selected free segment. (b) Garbage collection (write D) copies still-valid data to release unused space.

3. Garbage-collection process

To study the load imposed by garbage collection, we must start with a specific model of the garbage-collection process. A description is needed which is simple but at the same time provides enough flexibility to compare alternative collection strategies. To provide the needed model, we adopt a simplified version of the *age sort* strategy as proposed in [1].

The age sort scheme calls for collected data objects to be grouped into segments by age. This is done by sorting the set of objects which are in the process of being moved, then subdividing the sorted list into segments.

The *generation group* model tries to retain the spirit of the age sort, but avoids the analysis difficulties that arise from actually performing a sort. Rather than grouping by age, the generation group model instead organizes

collected objects according to the number of times they have previously been garbage-collected.

In the remainder of this section, we start by fleshing out the rough outline just given of the generation group model. We then examine what the model suggests about garbage-collection load and about the utilization of storage.

• *Generation group model*

In the generation group model, collected data objects are organized into segments on the basis of the number of times they have already been moved. This is done according to the following scheme:

- When a data object is first written, it belongs to generation 1; after the object is moved for the first time, it belongs to generation 2; and so forth. Thus, generation $i \geq 1$ consists of those data objects that have previously been moved exactly $i - 1$ times.
- Data objects are grouped into segments on the basis of their generation i.
- A steady state is assumed to exist in which a given segment is continually used and reused to store data objects from the same generation.
- There are $n \geq 1$ generation "groups." The generation group to which a data object in generation i belongs is given by $g = \min(i, n)$.
- Garbage-collection policies, such as the utilization threshold that triggers segment collection as discussed below, are established by generation group.

Note that generation group n lumps together all the generations $i \geq n$. Garbage-collection policies in this generation group do not take into account the actual generation of a given segment. Therefore, there are two important cases of the generation group model:

1. If $n = 1$, the segment-collection policy is history-independent.
2. If $n > 1$, the segment-collection policy is history-dependent.

For purposes of simplicity, we often boil our results down to the cases $n = 1$ (history-independent) and $n = 2$ (the simplest history-dependent case).

In the analysis of the generation group model, we consider the utilization of storage to be the independent variable in terms of which the garbage-collection load can be described. For the multiple-group case, there is a storage utilization for each individual group, and these must "add up" to the correct overall utilization.

Within a generation group, the selection of the next segment to collect is based on segment utilization; i.e., the least-occupied segments are collected first. By increasing or decreasing the utilization level at which a segment in a given generation group becomes a candidate for collection (the collection *threshold*), the utilization of the generation group can be raised or lowered. This can only be done, however, within the constraints imposed by the actual impact of the threshold on the utilization of the generation group and by the storage utilization of the subsystem as a whole. If the utilization of one group is reduced, the utilization of some other group must be increased in order to maintain a consistent overall utilization.

In the case of multiple generation groups, thresholds are used to achieve the desired combination of individual generation group utilizations, consistent with the constraints just described.

In the analysis which follows, we assume that no spare segments are held in reserve; i.e., all segments are in active use for data storage and belong to some generation $i = 1, 2, \cdots$. This assumption is made without loss of generality, since to analyze a subsystem with spare storage we need only limit the analysis to the subset of the storage that is actually in use. Note, however, that in a practical log-structured file at least a small buffer of spare segments *would* need to be maintained. This would reduce the utilization of storage that is achieved in practice to a level at least somewhat lower than that which we consider in the present analysis.

• *Analysis of a generation group*

We now turn to the application of the generation group model as a way to explore garbage-collection loads. Our starting point is to understand the collection activity and storage utilization of each individual generation group. We then extend these results into an analysis of the overall subsystem.

Collection activity

The collection activity of a given generation group $1 \leq g \leq n$ is understood in terms of the following variables:

- w = the rate per second at which new data objects are written for the first time to the disk medium after having been transferred from the host.
- w_g = the number of data objects per second being introduced into generation group g. Note that $w_1 = w$.
- b_g = the rate at which free storage in generation group g is reclaimed, expressed in terms of the number of data objects' worth of storage per second.
- c_g = the number of data objects per second being collected in generation group g. Although there is no generation group 0, we also define $c_0 = w$ for convenience in writing some of the formulas below.

s = the number of data objects' worth of storage available to the subsystem.

u = the average subsystem storage utilization (the actual number of valid data objects relative to s).

u_g = the average storage utilization of the segments belonging to generation group g.

f_g = the average utilization of segments being collected from generation group g. Also, the utilization threshold at which segment collection is triggered in group g.

$$= \frac{c_g}{b_g + c_g}.$$

When required, we also use the subscript i with some of the variables above, to indicate a specific generation i as opposed to a generation *group* g. The subscript n, or a numerical subscript 1, 2, etc., should always be taken to refer to the generation *group*; e.g., $g = n$ or $g = 2$. Care has been exercised in the context of any such usage to make the intended meaning clear.

We start our analysis by noting that, since new data objects being added to a given generation come only from the immediately preceding generation,

$$w_g = c_{g-1} \tag{1}$$

for all generation groups $1 \le g \le n$ (recall that, by convention, $c_0 = w$). Also, assuming steady-state operation, the rate at which storage is demanded for new data objects being added to any generation group must be the same as the rate at which storage is freed. In generations groups $g < n$, all storage in collected segments is freed, since any remaining data objects contained in these segments are passed to the next generation group. In generation group n, however, only the empty storage in collected segments is freed. Thus,

$$w_g = \begin{cases} b_g + c_g & 1 \le g \le n-1, \\ b_g & g = n. \end{cases} \tag{2}$$

By the definition of f, (2) means that

$$c_g = \begin{cases} f_g w_g & 1 \le g \le n-1, \\ \dfrac{f_g}{1-f_g} w_g & g = n. \end{cases} \tag{3}$$

Repeated application of (1) and (3) now yields the following formula for the garbage-collection activity in generation group g:

$$c_g = \begin{cases} f_g f_{g-1} \cdots f_1 w & 1 \le g \le n-1, \\ \dfrac{1}{1-f_g} f_g f_{g-1} \cdots f_1 w & g = n. \end{cases} \tag{4}$$

Storage utilization

To study storage utilization, it is necessary to sketch out the "life cycle" of a given data object. Let us therefore consider a time line for the object. The time line begins, at time 0, when the object is written by a host application.

Before the object is stored on the magnetic disk medium, it may be buffered. This may occur either in the processor (a DB2® deferred write, for example) or in the storage control. Let the average time at which the data object finally is written to the magnetic medium be called τ_0.

The data object is next packaged into a succession of one or more segments that contain various generations i, $i = 1, 2, \cdots$, of data objects. Let τ_i be the average time of collection of each such segment. Finally, the storage occupied by the object is released when it is rendered out of date. Let T_i be the average lifetime of those objects that become out of date while they are in generation i, and let T be the overall average lifetime of all data objects.

In our analysis later in the paper, it will be important to know how quickly data are written to disk relative to the overall lifetime of the data. We therefore define

$$d = \frac{\tau_0}{T}.$$

The ratio d tends to be of the same order as the amount of cache storage that contains "dirty" data, relative to the amount of disk storage. It typically ranges from zero (no buffering of writes) up to a few tenths of a percent.

To calculate the storage utilization of segments belonging to generation i, we apply Little's law. On one hand, the total number of data objects' worth of storage in these segments, as given by Little's law, must be $w_i(\tau_i - \tau_{i-1})$. On the other hand, the population of data objects still alive is

$$w_i\{f_i(\tau_i - \tau_{i-1}) + (1 - f_i)(T_i - \tau_{i-1})\},$$

since a fraction $1 - f_i$ of the objects are rendered invalid before being collected. We can therefore divide the number of live objects by the number of objects' worth of storage to obtain

$$u_i = f_i + \frac{T_i - \tau_{i-1}}{\tau_i - \tau_{i-1}}(1 - f_i). \tag{5}$$

Note that (4) and (5), taken together, dictate the relationship among garbage-collection load, storage utilization, and the actual workload running on the subsystem. The timing with which the workload causes data objects to be rendered out of date determines, via (5), the segment collection threshold that yields the desired storage utilization. The collection threshold, in turn, determines garbage-collection load via (4).

Equation (5) applies to individual generations; the results by generation must still be lumped together into the

required generation *groups*. Luckily, both of the workload models investigated later in the paper—the simple linear model, as well as the more realistic hierarchical reuse model—make the lumping into generation groups very easy because of the forms that (5) assumes in these models. We therefore defer the application of (5) to generation *groups* until it can be examined in the context of these specific workload models.

• *Analysis of the subsystem*

We now consider the analysis of the subsystem as a whole, making use of the results just obtained for each generation group.

Moves per write

The moves per write M for the subsystem as a whole are given by

$$M = \frac{1}{w} \sum_{g=1}^{n} c_g, \tag{6}$$

where the individual c_g are given by (4). For the purpose of this paper, there are two important special cases:

• For $n = 1$,

$$M = \frac{f_1}{1 - f_1}. \tag{7}$$

• For $n = 2$, (6) simplifies to

$$M = \frac{f_1}{1 - f_2}. \tag{8}$$

In either case, *the moves per write become unbounded as the segment utilization threshold approaches unity*.

Utilization

To understand overall subsystem storage utilization, we must now work out the precise meaning of the earlier statement that the storage utilizations of the individual generation groups must "add up" to the storage utilization of the subsystem as a whole.

Our starting point is to note that by Little's law, the total population of up-to-date data objects in the subsystem is Tw. But by the reasoning of the subsection on storage utilization, we can also write this population by adding up the contribution of each generation group:

$$Tw = u_n w_n(\tau_{n+} - \tau_{n-1}) + \sum_{g=1}^{n-1} u_g w_g(\tau_g - \tau_{g-1})$$

$$= u_n \left[w_n(\tau_{n+} - \tau_{n-1}) + \sum_{g=1}^{n-1} w_g(\tau_g - \tau_{g-1}) \right]$$

$$+ \sum_{g=1}^{n-1} (u_g - u_n) w_g(\tau_g - \tau_{g-1}).$$

However, again using Little's law, we find that the term in brackets is just s, the total number of objects' worth of storage. Since $us = Tw$, as discussed above, this means that

$$Tw = \frac{u_n}{u} Tw + \sum_{g=1}^{n-1} (u_n - u_g) w_g(\tau_g - \tau_{g-1}),$$

or, after rearrangement,

$$\frac{1}{u} = \frac{1}{u_n} + \sum_{g=1}^{n-1} \left(1 - \frac{u_g}{u_n}\right) \frac{w_g}{w} \left(\frac{\tau_g}{T} - \frac{\tau_{g-1}}{T}\right). \tag{9}$$

The constraint (9) must be met by any garbage-collection strategy that involves more than one generation group. In particular, the history-dependent scheme introduced in Section 5 uses this constraint to relate the utilizations in generation groups 1 and 2.

4. Linear model

We now make the results derived above more concrete by working them out in the case of a specific assumption about the pattern of updates. This assumption, the *linear model*, is chosen not for its realism but for its simplicity. Also for simplicity, we assume that there is only a single generation group.

Once the pattern of updates has been described, it is possible to relate garbage-collection load directly to the utilization of storage. It turns out that the general structure of the relationship that we are about to derive with the linear model stands up well, even when a more realistic update pattern is taken into account, as we do later in the paper.

The key assumption of the linear model is that the data objects in a given segment are superseded at a constant rate. A fixed number of objects per unit of time are rendered out of date from the moment that the segment is first filled until it is collected. This means that the average lifetime of an object superseded during this period is just

$$T_i = \frac{\tau_i + \tau_{i-1}}{2}. \tag{10}$$

From this it follows immediately by (5) that

$$u_i = f_i + \frac{1}{2}(1 - f_i) = \frac{1 + f_i}{2}.$$

The equation just stated gives the utilization for a specific generation i. But since this depends only upon f_i, and since

$$f_i = f_n \qquad i \geq n,$$

the same equation applies to generation groups as well as to individual generations:

$$u_g = \frac{1 + f_g}{2} \tag{11}$$

or

$$f_g = 2u_g - 1. \tag{12}$$

In the present analysis, there is exactly one generation group and $u_1 = u$ (since we have assumed that there are no spare segments). We can therefore plug (12) into (7) to obtain

$$M = \frac{2u - 1}{1 - (2u - 1)} = \frac{1 - 2(1 - u)}{2(1 - u)}$$

or

$$M = \frac{0.5}{1 - u} - 1. \tag{13}$$

This simple result provides an important point of comparison as we progress to the more realistic workload assumptions later in the paper. A plot of (13) is included in Figures 3–5 of the next section.

It should be noted that (13) is valid only for utilizations of at least 50 percent. For utilizations below 50 percent, the linear model predicts that at least some segments must necessarily play the role of spares (despite our earlier simplifying assumption), and that $M = 0$.

Considering, then, the range of storage utilizations to which it applies, (13) predicts that as the subsystem approaches 100 percent full, the garbage-collection load becomes unbounded. This conclusion continues to stand up in the light of our later, more detailed analysis. It suggests that a subsystem using log-structured files should not be pushed much above the range of 80 to 85 percent full.

5. Hierarchical reuse model

In the following section, we show that the *hierarchical reuse* model can be used to improve the realism and scope of the analysis just given. However, we do not much alter the qualitative conclusions just obtained from the linear model.

We begin by summarizing the update pattern predicted by the hierarchical reuse model and comparing these predictions to the actual patterns of updates seen in a number of traces. We then show how this model can be used to relate the utilization of each generation group to its collection threshold, in the same way that (11) does for the linear model. Finally, we examine the implications of this key relationship for both the case of a history-independent ($n = 1$) and a history-dependent ($n = 2$) garbage-collection algorithm.

- *Model description*

The hierarchical reuse model was first introduced in a previous paper [2] as a way to describe cache reference locality. A more complete study of the model is presented in that paper. We now summarize the model as it applies in the context of update activity.

The model's central assumption about updates is that they are caused by a series of hierarchically related processes. For example, repeated updates of a specific data object may occur within the same subroutine; within different routines called by the same transaction; or as part of an overall task that involves several transactions. Because of this hierarchical structure, we hypothesize that *the probability of data reuse at long time scales should mirror that at short time scales, once the time scale itself is taken into account.*

For example, consider the pattern of updates to the tracks on a 3380 or 3390 device. Imagine two tracks:

1. A *short-term* track, last updated five seconds ago.
2. A *long-term* track, last updated twenty seconds ago.

Hierarchical reuse probability, as applied to these two tracks, says that the short-term track has the same probability of being updated in the next five seconds as the long-term track does of being updated in the next twenty seconds; and that the short-term track has the same probability of being updated in the next minute as the long-term track does of being updated in the next four minutes.

To state this formally, let the random variable U be the time from the last update of a given track until the next update. Then we propose the following hierarchical reuse hypothesis:

> *The conditional distribution of the quantity*
>
> $$\frac{U}{u_0} \mid U > u_0$$
>
> *does not depend upon* u_0. *Moreover, this distribution is independent and identical across periods following different references.*

The behavior just hypothesized is a special case of *statistical self-similarity*, a feature often seen in the study of fractals. Indeed, it is shown in [5] that a random variable U which satisfies the conditions of the hypothesis above must belong to the *hyperbolic* family of

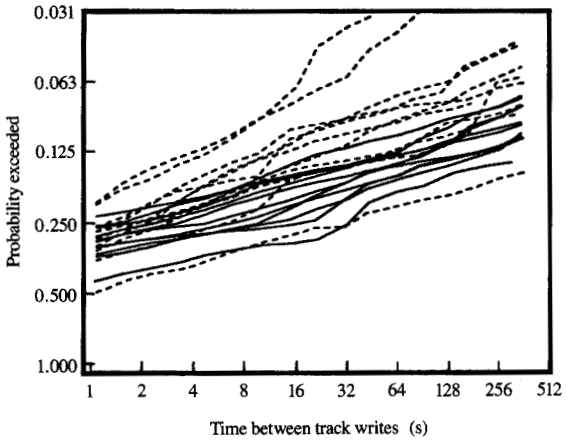

Figure 2

Distribution of time between track updates. User and system data pools are shown respectively as solid and dashed lines.

distributions. This means that for x large enough to exceed some lower limit on the time scale, U is characterized by

$$P[U > x] = ax^{-\theta} \tag{14}$$

for some constants a and θ.

If update activity follows the example of most real processes that exhibit fractal behavior, we must expect both a lower limit, as just mentioned, and an upper limit on the applicable time scale. In the case of update patterns, the lower limit appears to be much less than any time period of interest (less than one second). Unfortunately, there does not appear to be any good way to estimate the upper limit on the basis of currently available data. Clearly, however, (14) *cannot* apply at time scales beyond of the life of the device.

If the probability predicted by (14) is shown as a function of x in a plot with log–log axes, the result is a straight line. **Figure 2** presents a check of this prediction against the disk volumes observed in eleven VM installations as part of the survey reported in [4]. All volumes at each installation were divided into rough groups, or "pools," depending upon the predominant type of data contained on the volume. The figure plots both the user data (general user and database storage) and the "KEY SYS" data as observed at each installation.

Figure 2 comes strikingly close to being the predicted collection of straight lines. Therefore, (14) *can* be accepted as a realistic description of update patterns. Unfortunately, as discussed above, Figure 2 does not provide enough information to set an upper limit on the applicable time scale. In this paper, we therefore make a "leap of faith," and assume that (14) applies at least up to the time scales that are relevant to garbage collection (from a few minutes to several weeks or months).

Figure 2 also helps with the problem of what values of θ to assume (at least for data objects that are track images). The value of θ is dictated by the slope that appears on the plot. With remarkable consistency, the slopes presented in Figure 2 fall into the range $0.2 \le \theta \le 0.3$. We therefore adopt $\theta = 0.25$ as a "middle-of-the-road" value to use in evaluating some of the model results below. We also consider the values $\theta = 0.2$ and $\theta = 0.3$ in order to explore the sensitivity of these results with respect to θ.

As Figure 2 makes clear, there is a strong tendency for updates to exhibit *locality*—i.e., updates to a given data object tend to be closely spaced in time. This behavior contrasts sharply with the constant rate of invalidation assumed by the linear model. The vital contribution of the hierarchical reuse model to the analysis of garbage-collection loads is that it provides a way to account for such locality. We now examine how our earlier results can be refined by applying the hierarchical reuse model.

Storage group utilization

The calculation of storage group utilizations based on (14) is performed by applying exactly the same steps as for the equivalent calculation based on the linear model. First, we obtain the average lifetime of data objects that are superseded while in generation i. The average lifetime is then plugged into (5).

In the case of the hierarchical reuse model, these calculations are too lengthy to incorporate as part of the paper. Eventually, however, they yield the simple and interesting result

$$u_g = \frac{1}{1-\theta}\,\frac{f_g - f_g^{1/\theta}}{1 - f_g^{1/\theta}}. \tag{15}$$

Note that this equation shares with the corresponding equation (11) of the linear model the property that the storage utilization depends upon the collection threshold in the same way regardless of the specific generation being examined. As in the case of the linear model, this property means that (15) applies equally well both to individual generations and to generation groups.

The utilization given by (15) also has an interesting *quantitative* relationship to the corresponding linear result. This is made clear by writing the second-order expansion of (15) in the neighborhood of $f_g = 1$:

$$u_g \simeq 1 - \frac{1-f_g}{2} - \frac{1+\theta}{12\theta}(1-f_g)^2$$

$$= \frac{1+f_g}{2} - \frac{1+\theta}{12\theta}(1-f_g)^2. \tag{16}$$

This gives a practical approximation for values of f_g greater than about 0.6. As a comparison of (16) and (11) suggests, the utilization predicted by the hierarchical reuse model is always less than that given by the linear model, but the two predictions come into increasingly close agreement as the collection threshold approaches unity.

- *History-independent collection*

Consider, now, the garbage-collection activity for a history-independent garbage-collection algorithm ($n = 1$). In this case, there is only one generation group. For any value of f_1, the moves per write in this single generation group are given by (7). The utilization of the single generation group is given by (15). By examining a succession of values f_1, moves per write can therefore be plotted against storage utilization.

Figure 3 presents the results of several such plots, showing a range of values θ. Also shown, for comparison, is the prediction given by the linear model. As Figure 3 shows, the performance of the history-independent model is not strongly dependent on the exact value of θ within the range of this parameter that seems to be of interest. However, the hierarchical reuse model does increase our estimate of the moves per write compared to that obtained from the linear model. Remarkably, the actual number of moves per write by which the estimate increases is roughly constant across the entire range of utilizations from 50 to 95 percent and for all three plotted values of θ. Thus, we end up with the following simple approximation of the hierarchical reuse result:

$$M \simeq \frac{0.5}{1-u} - 0.3. \tag{17}$$

- *History-dependent collection*

Putting the result just stated into a nutshell, we have found that the presence of update locality creates a problem for history-independent garbage collection. Essentially, the reason is that the rate of rendering data objects in a segment out of date slows down as the segment ages. This increases the longevity of empty storage that becomes out of date early in the life of the segment, and reduces the utilization that can be achieved in any given generation group at any given cost in terms of moves per write.

Update locality also creates an opportunity to improve the performance of garbage collection, however. Because of locality, data objects have a substantial probability of becoming out of date within a short time after they are written by the host. This suggests delaying collection of a segment that contains recently written objects until the segment is mostly empty. Through such a delay, it may be possible to avoid ever moving objects that have short lifetimes.

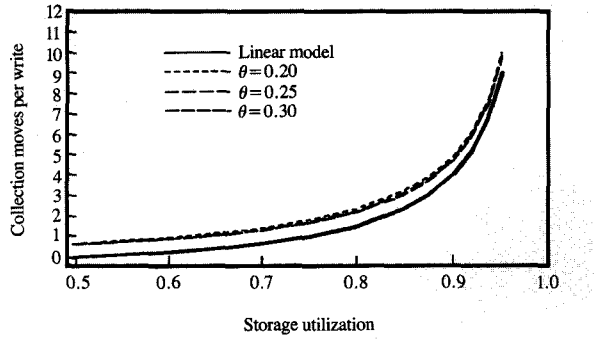

Figure 3

Results for history-independent garbage collection.

Such a delay can only be practical if it is limited to recently written data; segments containing older data would take too long to empty because of the slowing rate of invalidation. Therefore, a history-dependent garbage-collection strategy is needed to implement this idea. In the following subsection, we experiment with the strategy just outlined by examining a garbage-collection scheme with two generation groups: group 1 to hold recently written data, and group 2 for older data.

In the hierarchical reuse model, the values of τ_i depend upon the quantities f_i due to (14):

$$\tau_i = f_i^{-1/\theta}\tau_{i-1} \qquad i = 1, 2, \cdots, \tag{18}$$
$$\tau_0 = dT.$$

Thus, for the case $n = 2$ we can rewrite (9) as

$$\frac{1}{u} = \frac{1}{u_2} + d\left(1 - \frac{u_1}{u_2}\right)(f_1^{-1/\theta} - 1). \tag{19}$$

To define a garbage-collection scheme with two generation groups, we must specify the values of four variables: f_1, f_2, u_1, and u_2. These variables must satisfy (15) as it applies to each of the two generation groups, and must also satisfy (19). Within these constraints, they must also produce the smallest possible number of moves per write, as given by (8). We are confronted, therefore, by a minimization problem involving four unknowns, three equations, and an objective function.

To explore the history-dependent strategy, iterative numerical techniques were used to perform the minimization just described. This was done for a range of storage utilizations, values θ, and ratios $d = \tau_0/T$. The results of the iterative calculations are presented by

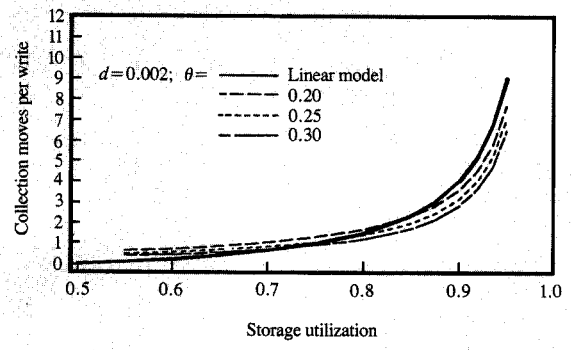

Figure 4

History-dependent moves per write: sensitivity to θ.

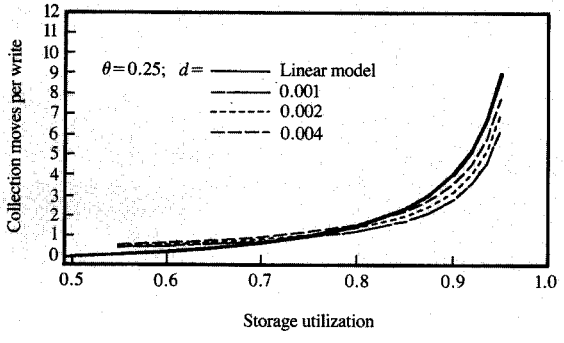

Figure 5

History-dependent moves per write: sensitivity to d.

Figures 4 and **5**. For the purpose of these figures, the case $\theta = 0.25$ and $d = 0.002$ is used as a starting point. Figure 4 varies θ around this case, while Figure 5 varies d.

The two figures show clearly that history-dependent garbage collection performs better than history-independent garbage collection as presented in Figure 3. The amount of improvement depends upon the parameters θ and d, as well as on the utilization of the subsystem.

The potential improvement is most dramatic at utilizations higher than 85 percent. This operating range may be too high for normal use, however, because of the very rapid increase in moves per write that occurs for utilizations over 85 percent.

If utilizations in the range of 75 to 85 percent are considered, most of the results presented in Figures 4 and 5 appear to be at or a little below the number of moves per write predicted by the linear model. For these storage utilizations, the linear model, as given by (13), appears to provide a serviceable "rule of thumb."

6. Summary and discussion

The result just stated brings us full circle. Initially the estimate

$$M = \frac{0.5}{1-u} - 1 \qquad 0.5 \le u < 1$$

was introduced on the grounds that it represented the simplest possible assumption about the pattern of updates in the workload. We have now concluded that as a "rule of thumb," this estimate comes fairly close to the truth for environments with the scattered updates typical of database I/O—even after accounting for a much more realistic model of the pattern of updates and after applying a sophisticated, history-dependent garbage-collection algorithm.

Our other key conclusion is that in such environments, efficient operation of a log-structured disk subsystem demands that no more than roughly 80 percent of available storage capacity must be utilized (minus at least a small buffer of reserve segments, as discussed in the subsection on the generation group model). So long as storage utilization is kept at this level or below, however, and so long as an effective history-dependent collection policy is adopted, log-structured subsystems should be capable of the high write throughput pointed to by [3] as the motive for the concept. Nevertheless, garbage collection *will* affect the throughput achieved, by requiring an amount of data movement that is quantified roughly by the above rule of thumb.

It should be noted that traditional disk subsystems must also ensure that some free storage is available, so that it is always possible to allocate storage for new files (the amount of such storage depends upon the size of ongoing file allocations). Also, since a log-structured subsystem writes all data objects into fresh segments, it can deal easily with variations in data object size when updating an old copy of a given data object. Such a subsystem may therefore be capable of providing automatic data compression [7, 8]. Thus, despite the effective loss of storage capacity needed to ensure efficient operation, log-structured disk subsystems may still, on balance, improve upon the amount of data that can be stored on disk by traditional subsystems.

It may be useful to conclude by considering the use of log-structured files to supplement a RAID-5 disk parity architecture. This scheme has recently been the focus of much attention because of its introduction into the high-end disk marketplace.

The concept of a disk subsystem architecture using Redundant Arrays of Independent Disks (RAID) was proposed in 1988 by Patterson, Gibson, and Katz [9]. The key to the concept is the use of one or more disks to contain enough redundant data to reconstruct the contents of any disk should one fail. Several variations are defined in [9], ranging from mirrored disks (RAID-1) to the sophisticated rotating parity scheme of RAID-5. For database or transaction-processing workloads, however, RAID-5 appears to be the most attractive of these variations.

In its standard form, the RAID-5 scheme calls for several disks to be formed into a "parity group." Corresponding locations of all parity group disks except one contain stored data. The same location on one, selected disk contains redundant (parity) information needed for data reconstruction in the event of any single disk failure. The various disks of the parity group rotate the job of carrying the parity information for a given corresponding location across the parity group, so as to avoid placing too much load on any one disk.

The RAID-5 scheme for disk parity requires four operations in order to carry out a write requested by the host [9]: read data, read parity, write data, and write parity. This is necessary so that the parity disk can correctly account for the contents of all disks in the parity group, including those not affected by the write. The fact that four operations are needed per data object written, compared with one operation in the case of a standard disk subsystem, is called the RAID-5 *write penalty*.

If a write is performed simultaneously to all the disks of a RAID-5 parity group, by contrast, the storage control needs no information from disk to set parity correctly. Only a single operation is needed per data object written, plus one write for parity update that can be overlapped with the writes of the data objects.

It has sometimes been argued, for this reason, that by building a log-structured file system on top of the RAID-5 architecture it is possible to eliminate the write penalty. To accomplish this, each segment is defined so that it includes corresponding areas on all disks of a parity group. Therefore, all write operations are performed across a full parity group and are free from the write penalty.

As we have seen, however, there is a complication inherent in this idea. Although the write operation *itself* can be performed without penalty, a segment's worth of free storage must first be garbage-collected. The number of operations per write needed for garbage collection will depend, as described in the earlier sections of the paper, upon the workload and the utilization of storage.

Let us consider, then, the number of operations required for the middle case of Figure 4, at various storage utilizations (where each read or write of a data object is counted as one "operation"):

- At a utilization of 70 percent, we require 0.87 moves per write. Each move requires one read and one write; so adding in the initial write of the object we get 2.74 operations. This is a substantial improvement over the four operations required by the standard RAID-5 design.
- At a utilization of 80 percent, we require 1.42 moves per write. This gives 3.84 operations per write, about the same as RAID-5. (The rule-of-thumb estimate would yield four operations per write, exactly the same as RAID-5, for this utilization.)
- At a utilization of 90 percent, we require 3.16 moves per write. This gives 7.32 operations per write, almost double the requirements of RAID-5.

In making comparisons of this kind, it is important (but not easy) to consider the efficiency of the operations. If reads and writes are performed on a segment-wide basis, and are not interrupted, a high efficiency can be achieved per data object read or written in the segment. On the other hand, segment-wide reads and writes for garbage collection *should* be interruptible because of their background nature and their long times to completion. At high subsystem loads, interruptions would be frequent and would create the potential for large amounts of waste motion. Presumably, however, this effect can be controlled so as to ensure reasonable efficiency of background operations under high loads.

In any case, it is clear that we *cannot* eliminate the write penalty of RAID-5 just by using a log-structured subsystem. We can only change its form. At storage utilizations in the range of 80 percent, the use of a log-structured subsystem seems to have little effect one way or the other on the RAID-5 requirement of four operations per write. With suitable scheduling of background work, such files may, however, improve the efficiency with which it is possible to carry out the required operations.

References

1. M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Computer Syst.* **10**, No. 1, 26–52 (1992).
2. M. Rosenblum, "The Design and Implementation of a Log-Structured File System," *UCBCS Technical Report 92/696*, University of California at Berkeley, 1992.
3. J. Ousterhout and F. Douglis, "Beating the I/O Bottleneck: A Case for Log-Structured File Systems," *ACM Oper. Syst. Rev.* **23**, No. 1, 11–28 (January 1989).
4. B. McNutt, "A Simple Statistical Model of Cache Reference Locality, and Its Application to Cache Planning, Measurement, and Control," *Proceedings of CMG*, December 1991, pp. 203–210.

5. B. B. Mandelbrot, *The Fractal Geometry of Nature*, Revised Edition, W. H. Freeman & Co., New York, 1983. See particularly p. 383.
6. B. McNutt, "An Overview and Comparison of VM DASD Workloads at Eleven Installations, with a Study of Storage Control Cache, Expanded Storage and Their Interaction," *Proceedings of CMG*, December 1989, pp. 306–318.
7. M. Burrows, C. Jerian, and B. Lampson, "On-Line Data Compression in a Log-Structured File System," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems,* September 1992, Vol. 27, No. 9, pp. 2–9.
8. A. L. Jones, "The Implications and Potentials of Advanced Disk-Array Technology," *Computer Measurement Group Trans.*, Issue No. 79, 59–63 (Winter 1993).
9. D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of ACM SIGMOD 88*, June 1988, pp. 109–116.

Received September 3, 1993; accepted for publication January 26, 1994

Bruce McNutt *IBM Storage Systems Division, 5600 Cottle Road, San Jose, California 95193 (MCNUTT at SJEVM13)*. Mr. McNutt is a senior engineer-scientist working in DASD system performance evaluation. He joined IBM in 1983 and has specialized in DASD performance and workload characterization since that time. His 1990 paper "DASD Configuration Planning: Three Simple Checks" received the award for Best Management Paper at that year's conference of the Computer Measurement Group (CMG). In 1987 he originated the "multiple workload" approach to planning for storage control cache memory (planning based upon an objective for cache holding time), which is now in widespread use. Mr. McNutt received his B.S. degree in mathematics from Stanford University and his master's degree in electrical engineering and computer science from the University of California at Berkeley.