Semi-automatic, Self-adaptive Control of Garbage Collection Rates in Object Databases

Jonathan E. Cook
University of Colorado
jcook@cs.colorado.edu

Artur W. Klauser
University of Colorado
klauser@cs.colorado.edu

Alexander L. Wolf
University of Colorado
alw@cs.colorado.edu

Benjamin G. Zorn
University of Colorado
zorn@cs.colorado.edu

Abstract

A fundamental problem in automating object database storage reclamation is determining how often to perform garbage collection. We show that the choice of collection rate can have a significant impact on application performance and that the "best" rate depends on the dynamic behavior of the application, tempered by the particular performance goals of the user. We describe two semi-automatic, self-adaptive policies for controlling collection rate that we have developed to address the problem. Using trace-driven simulations, we evaluate the performance of the policies on a test database application that demonstrates two distinct reclustering behaviors. Our results show that the policies are effective at achieving user-specified levels of I/O operations and database garbage percentage. We also investigate the sensitivity of the policies over a range of object connectivities. The evaluation demonstrates that semi-automatic, self-adaptive policies are a practical means for flexibly controlling garbage collection rate.

1 Introduction

Automatic storage reclamation, or *garbage collection* (GC), is becoming recognized as an important new feature for object database management systems (ODBMSs). A number of recent research papers have considered some of the important aspects of the correctness and performance of ODBMS garbage collection [AFG95, CWZ94, KLW89, KW93, YNY94]. A recently proposed standard suggests using garbage collection for at least some of the programmatic interfaces to an ODBMS [Cat93]. Commercial ODBMSs are now providing implementations of garbage collection (e.g., [Cor94]).

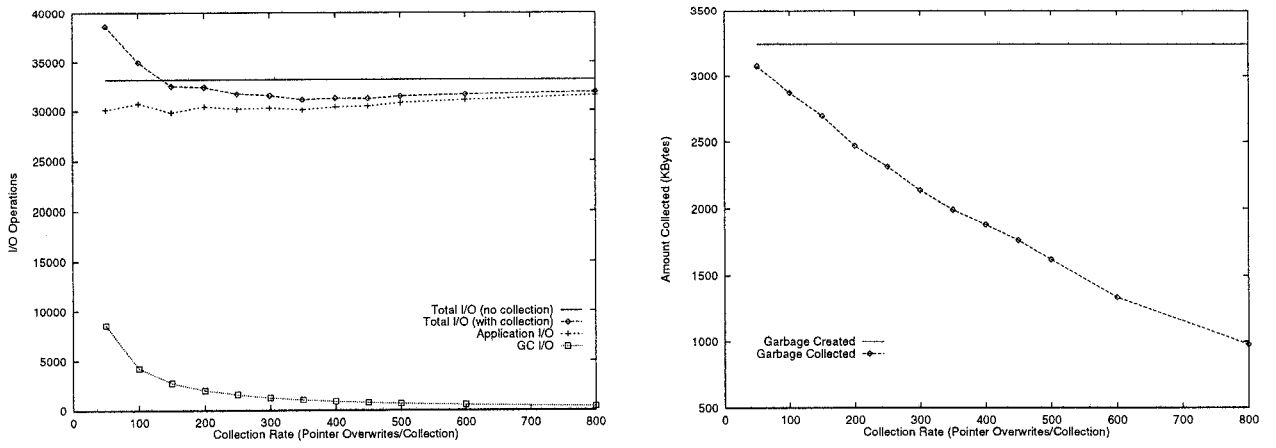
This work was supported in part by the National Science Foundation under grant IRI-95-21046. A. Klauser was supported by a BMfWF/Fulbright Graduate Fellowship.

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

In a previous paper [CWZ94] we presented a framework for investigating the issues surrounding partitioned garbage collection of ODBMSs. Partitioned collection is an incremental technique based on manipulating disjoint portions of a database [YNY94] and is akin to generational collection in programming language systems [Wil92]. We categorized the issues into a number of policy areas that together contribute to a complete garbage collection algorithm. We described the results of our investigation in one policy area, *partition selection*, which is the selection of which partition of a database to collect during a given garbage collection. In that paper we introduced a new partition selection policy, called UPDATEDPOINTER.

In this paper we investigate another critical policy area of partitioned garbage collection algorithms, that of determining how often to perform garbage collection. We refer to this policy area as the *collection rate*. Intuitively, we can understand how collection rate impacts both I/O performance and database size. If garbage collection occurs frequently, then the number of I/O operations associated with reclamation will dominate the number of I/O operations associated with the application, but very little garbage will exist in the database. Conversely, if collection occurs infrequently, then the impact of reclamation on I/O performance will be small, but a significant amount of garbage may accumulate in the database between collections, reducing storage efficiency and possibly increasing access time. Thus, finding an appropriate collection rate is an exercise in determining a time/space tradeoff between I/O and storage overheads.

Figures 1a and 1b show the effect of varying the collection rate on the I/O performance and on the total garbage collected in a test database. Specific details of the test database, an instance of the OO7 benchmark [CDN93], are discussed in Section 3.3. The figures highlight the time/space tradeoff of collection rate policies. For example, it is clear that choosing a collection rate for this application of "50", measured in pointer overwrites (i.e., modifications of pointers between objects) per collection, results in excessive



(a) (b)

Figure 1: Collection Rate versus I/O Operations (a) and Total Garbage Collected (b).

numbers of I/O operations, while choosing a collection rate of "800" pointer overwrites per collection results in little garbage being collected.[1] So the question remains, what is a reasonable rate for collecting garbage?

Unfortunately, this question depends on a number of different parameters and, as a result, is difficult to answer in general. Foremost, there is the issue of the relative importance of I/O overhead versus storage overhead. This issue is best decided by the ODBMS user,[2] which is tantamount to saying that the choice of a collection rate achieving the desired optimization is necessarily application dependent. Thus, it is inappropriate for the ODBMS implementor to preset a collection rate. On the other hand, in order for the ODBMS user to determine a suitable collection rate, they would need to understand the performance of the application as a function of collection rate by gathering data similar to those of Figure 1. Of course, for any significant database, such an exploration of application performance is costly in execution time and in human effort. Moreover, the data would reflect just that single application, which may be in conflict with other applications manipulating the same database.

A reasonable conclusion, therefore, is that a mechanism for controlling collection rate should be *semi-automatic*, in that the ODBMS, rather than the ODBMS implementor or the ODBMS user, should set the rate. More than that, the mechanism should be *self-adaptive*, in that the collection rate can vary in response to the dynamic behavior of the applications manipulating the database. Adaptiveness has two components: *responsiveness*, which is the speed with which a mechanism recognizes a change in behavior and reacts to that change, and *accuracy*, which is the degree to which the mechanism correctly responds to the change. The role of the ODBMS implementor and user should be to provide the broad performance goals that implicitly guide the ODBMS's semi-automatic, self-adaptive control of the collection rate.

In this paper we describe and evaluate two new ODBMS policies for determining appropriate collection rates. The policies are given input from the ODBMS user about the relative importance of I/O or storage resources and are adaptive to the dynamic behavior of database applications. In particular, our policies allow the user to specify a target percentage of I/O operations to be dedicated to garbage collection or a target percentage of garbage to be allowed to exist in the database. For example, if the specified target I/O percentage is 5%, then the collector automatically adjusts the collection rate to match the total number of garbage collection I/O operations to that percentage. As the mix of I/O operations changes, the collection rate adjusts to maintain the target percentage.

The policies we describe are intended to provide an exact level of performance (e.g., garbage or I/O percentage), under the assumption that the levels defined by the user apply while the database is under an active workload. When, for example, the database is quiescent, it would be desirable to have the collector run beyond its user-stated limits. In the case of I/O operations, this means a higher percentage of I/O can be given over to the collector in order to reduce the garbage in the database, while in the case of garbage percentage, this means the collector could attempt to reduce the amount of garbage below what was requested. Although we do not explore this dimension here, our policies can be extended to handle such situations.

We present a performance evaluation of our two

[1] As we explain in Section 2, one meaningful measure of collection rate is in terms of the number of pointer overwrites per collection.

[2] By "user" we mean the database administrator, the application developer, or the application user.

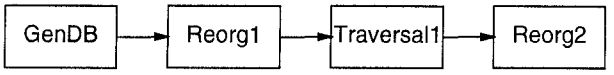


Figure 2: Phases of the OO7 Test Application.

new collection rate policies that is based on trace-driven simulations of an application developed by Yong, Naughton, and Yu [YNY94] for the OO7 ODBMS benchmark [CDN93]. The application consists of four distinct phases, each exhibiting behavior different from the one that it precedes. Figure 2 illustrates the progression of the phases, which we describe fully in Section 3. Our results show that the policies give excellent performance, accurately and responsively reacting to changes in application behavior. Furthermore, our collection rate policies add only little time and space overhead. We also show that our results hold across databases with different object connectivities.

While there has been a significant amount of research in object database garbage collection [AFG95, Bjö93, But87, CA86, CWZ94, KLW89, KW93, ML94, Mat85, YNY94], none of that previous work has investigated the issue of collection rate. For example, Yong, Naughton, and Yu propose a partitioned garbage collection policy, but assume that collection is triggered either when free-space becomes unavailable or after a fixed amount of storage is allocated [YNY94]. Their concern in that work is in comparing partitioned garbage collection against other approaches, and they choose a collection rate policy drawn from the realm of programming languages. In contrast, our work is aimed at both quantifying the cost of poor collection rate choices and proposing new policies for effectively controlling the collection rate in object databases.

There are a number of similarities between *copying* garbage collection [Che70] and on-line object *reclustering*. In particular, they both have the potential to improve application performance by relocating objects, they both incur additional execution overhead that must be balanced against the performance benefits they provide, and they both must find an appropriate rate at which to operate effectively. In recent work, for example, McIver and King [WJMK94] investigate the performance of an algorithm in which on-line reclustering is triggered when a measure of reference locality (the "external tension") exceeds a certain threshold and when a cluster analysis determines that reclustering might improve performance. The goal of their work is strictly to reduce the total number of application I/O operations. Our work is complicated by the additional (and sometimes conflicting) goal of trying to reduce the amount of storage required by the database.

Both reclustering and garbage collection can be addressed via off-line algorithms or "opportunistic" on-line algorithms that attempt to schedule large-scale database reorganizations when there is little database activity. Because many databases currently cannot be taken off-line and seldom have periods of low activity, we feel that investigating algorithms for on-line reclustering and garbage collection remains very important.

In Section 2 we present our new collection rate policies. Section 3 describes the experimental method we use to evaluate the policies and details the test databases and application we use in the evaluation. Section 4 presents the results of our experiments and Section 5 summarizes our findings.

2 Collection Rate Policies

In order to reclaim the most garbage, we need to identify what events occur that indicate when garbage is created. In programming language GC algorithms, object allocation and garbage creation are often assumed to be correlated. As a result, a heuristic of collecting after allocating a fixed number of bytes is sometimes used. However, allocation and garbage creation are not always correlated in object databases. Alternatively, we know that when pointers are overwritten, the objects are "less connected". Overwriting the final pointer to an object or group of objects actually does create garbage. Thus, we choose to use pointer-overwrite events as an indicator that garbage is being created in the database.

2.1 Fixed Rate Policies

A very simple collection rate policy is one that fixes the rate of collection over all applications. Unfortunately, a policy that chooses a fixed rate is destined to fail, as the amount of collection required will vary from database application to application, and any particular choice will not be optimal for all applications.

A more clever fixed-rate policy might attempt to determine the collection rate based on application characteristics, such as connectivity, object size, and ODBMS characteristics, such as partition size. For example, we know that the OO7 application we use has an approximate average connectivity of four (i.e., each object has four pointers pointing to it), and that object size is 133 bytes on average. From this, we could infer that every four pointer overwrites creates 133 bytes of garbage. If we assume that partitions are 96 kilobytes in size, then an obvious choice for collection rate would be to collect every 2956 pointer overwrites—that is, when a partition's worth of garbage has been created. Unfortunately, this simple heuristic also fails miserably. The OO7 application actually creates garbage at a rate of one kilobyte per six pointer overwrites, or *five times* more garbage than the simple calculation would predict.

There are two reasons why such simple heuristics fail. First, some individual overwrites can detach large

connected structures from the rest of the database and the heuristic does not capture this possibility. Second, a single overwrite may disconnect very large objects from the database, such as OO7 document nodes.

Another failing of fixed-rate policies is that they cannot adapt to changes in the database behavior. The OO7 application we use, for example, has two distinct reorganizations with very different properties. As a result, any fixed-rate policy used in this application will fail to work effectively for one or the other reorganization. We conclude that fixed-rate policies are unacceptable.

The obvious alternative to a fixed-rate policy is a policy that adjusts the collection rate automatically in an effort to achieve an optimal result. Unfortunately, because a time/space tradeoff is involved, there is no global "optimum" to achieve. As a result, we have investigated two semi-automatic policies that control collection rate based on user preferences. The first policy attempts to limit garbage collector I/O operations and we call it the *Semi-Automatic I/O* (SAIO) policy. The second policy attempts to limit the amount of garbage in the database, and we call it the *Semi-Automatic GArbage* (SAGA) policy. Both of these policies are self-adaptive, that is, they adjust the collection rate dynamically as the database application behavior changes. We use methods from control theory to develop the policies. To our knowledge, control theory has not been previously applied to this problem.

2.2 SAIO: An I/O Percentage Policy

Under the SAIO policy, the database user indicates what fraction of I/O operations should be used to perform garbage collection. For example, if the user wants garbage collection to utilize approximately 10% of the total I/O operations, then the user would set the SAIO parameter (called *SAIO_Frac* below) to 10%. In this policy, we use a count of I/O operations as a unit of time, because it corresponds exactly to the value the policy is trying to control.

The count of I/O operations indicates a behavior of the system, not a state. This means that I/O operation counts are always coupled with a measurement period, which we refer to as a *history* below. A natural time period in our application is the time between two successive collections. We use integral numbers of this basic period to describe history in our formulation of the SAIO rate policy. To describe a history, we use the notation $x|_a^b$ to indicate the history of x starting at a and ending at b. To express the history from the current collection to the next collection we use the intuitive notational abbreviation $\Delta x \equiv x|_c^{c+1}$.

In the formulation of the SAIO policy, we use the following definitions.

$SAIO\_Frac$	... requested collector I/O percentage
$AppIO$	... application I/O operations
$GCIO$	... garbage collection I/O operations
c	... current collection
c_{hist}	... history size (number of collections)

The goal of the SAIO policy is to determine the interval $\Delta AppIO$ after which the next garbage collection should occur, such that the *SAIO_Frac* constraint is met. In terms of I/O histories, this can be formulated as follows.

$$\begin{aligned} GCIO|_{c-c_{hist}+1}^{c+1} &= GCIO|_{c-c_{hist}+1}^{c} + \Delta GCIO \\ AppIO|_{c-c_{hist}+1}^{c+1} &= AppIO|_{c-c_{hist}+1}^{c} + \Delta AppIO \\ GCIO|_{c-c_{hist}+1}^{c+1} &= AppIO|_{c-c_{hist}+1}^{c+1} * SAIO\_Frac \end{aligned}$$

$$\begin{aligned} &GCIO|_{c-c_{hist}+1}^{c} + \Delta GCIO \\ &\quad = (AppIO|_{c-c_{hist}+1}^{c} + \Delta AppIO) * SAIO\_Frac \\ \Delta AppIO &= \frac{GCIO|_{c-c_{hist}+1}^{c} + \Delta GCIO}{SAIO\_Frac} - AppIO|_{c-c_{hist}+1}^{c} \\ &\approx \frac{GCIO|_{c-c_{hist}+1}^{c} + CurrGCIO}{SAIO\_Frac} - AppIO|_{c-c_{hist}+1}^{c} \end{aligned}$$

where $CurrGCIO \equiv GCIO|_{c-1}^{c}$. The approximation for $\Delta AppIO$ is achieved by making the assumption that $\Delta GCIO = CurrGCIO$, which means successive garbage collections incur approximately the same number of I/O operations.

To implement this policy, the collector must be able to determine the number of application and collection I/O operations. Additionally, the ODBMS must be able to trigger a garbage collection after the calculated number of application I/O operations has occurred. Since ODBMSs typically perform I/O operations explicitly, this requirement does not pose a problem.

2.3 SAGA: A Garbage Percentage Policy

Under the SAGA policy, the database user indicates what fraction of the database should contain garbage. For example, if the user wants garbage to account for approximately 5% of the total database size, then the user would set the SAGA parameter (called *SAGA_Frac* below) to 5%.

In the formulation of the SAGA policy, we use the following definitions.

$SAGA\_Frac$	... requested garbage percentage
$DBSize(t)$	... total database size by time t
$TotGarb(t)$	... total garbage generated by time t
$TotColl(t)$	... total garbage collected by time t
$ActGarb(t)$	... actual database garbage by time t = $TotGarb(t) - TotColl(t)$
$TargetGarb(t)$	... target database garbage by time t = $DBSize(t) * SAGA\_Frac$
$GarbDiff(t)$	... $ActGarb(t) - TargetGarb(t)$
$CurrColl$	... amount of garbage collected by the current collection

If *CurrColl* amount of garbage is collected at time t, the policy makes the assumption that approximately the same amount of garbage will be collected during the next collection. The policy also assumes that the database size will not grow significantly between t and $t + \Delta t$. Thus, $TargetGarb(t) \approx TargetGarb(t + \Delta t)$. Solving for Δt, we get the following.

$$\begin{aligned} ActGarb(t+\Delta t) &= TotGarb(t+\Delta t) - TotColl(t+\Delta t) \\ &= TotGarb(t+\Delta t) - (TotColl(t) \\ &\quad + CurrColl) \\ TotGarb(t+\Delta t) &= TotGarb(t) + TotGarb'(t) * \Delta t \\ ActGarb(t+\Delta t) &= TotGarb(t) + TotGarb'(t) * \Delta t \\ &\quad - (TotColl\ (t) + CurrColl) \\ &= ActGarb(t) + (TotGarb'(t) * \Delta t) \\ &\quad - CurrColl \end{aligned}$$

With the assumption of insignificant database growth between successive collections, our goal is then the following.

$$ActGarb(t+\Delta t) = TargetGarb(t+\Delta t) \approx TargetGarb(t)$$

We can then simplify and solve for Δt.

$$\begin{aligned} TargetGarb(t) &= ActGarb(t) + (TotGarb'(t) * \Delta t) \\ &\quad - CurrColl \\ CurrColl &= (ActGarb(t) - TargetGarb(t)) \\ &\quad + (TotGarb'(t) * \Delta t) \\ CurrColl &= GarbDiff(t) + (TotGarb'(t) * \Delta t) \\ \\ \Delta t &= \frac{(CurrColl - GarbDiff(t))}{TotGarb'(t)} \end{aligned}$$

To implement this policy, the ODBMS must estimate $TotGarb'(t)$, which is the slope of the *TotGarb* function. Thus, the collector must maintain some history information about previous estimates of the total garbage in order to estimate how much garbage will occur in the future. We estimate $TotGarb'(t)$ using a simple formula. Given a previous slope estimate, $TotGarb'(t_{prev})$, a previous pair of data points (t_{prev}, $TotGarb(t_{prev})$), and a current set of data points (t, $TotGarb(t)$), we estimate:

$$\begin{aligned} TotGarb'(t) &= Weight * TotGarb'(t_{prev}) \\ &= + \frac{(1 - Weight)(TotGarb(t) - TotGarb(t_{prev}))}{t - t_{prev}} \end{aligned}$$

where *Weight* is a controlling factor that buffers the policy from rapid changes in slope. We currently set $Weight = 0.7$. Also note that in practice, Δt can become very large if $TargetGarb'(t)$ approaches zero, or even negative. As a result, we place a minimum ($\Delta t_{min} = 2$) and maximum ($\Delta t_{max} = 1000$) on the value of Δt. We find that our policy works well in practice and that Δt_{min} and Δt_{max} are rarely utilized by the policies (e.g., see Figure 7 in Section 4).

2.4 Garbage Estimation for the SAGA Policy

One major difference between the SAIO policy and the SAGA policy is that the information needed to compute Δt for SAGA is not readily available. In particular, *ActGarb(t)* cannot be determined without scanning the entire database. As a result, to implement the SAGA policy practically, heuristics must be employed that can estimate the current amount of garbage in the database. We have invented and investigated several such heuristics, two of which we describe below. To help evaluate them, we have implemented in our simulator a perfect garbage "estimator" that knows exactly how much garbage exists in the database.

Garbage estimation can be split into two orthogonal components.

1. *State.* Describes the potential amount of garbage in each partition. Depending on the granularity, we differentiate between coarse and fine grain state descriptions. Coarse grain state (CGS) characterizes the database simply as the number of allocated partitions, whereas fine grain state (FGS) characterizes the database in terms of the number of pointer overwrites in each partition, which is based on the observation that pointer overwrites highly correlate with garbage creation. During a collection, the FGS value of one single partition changes from x to 0—that is, all potential garbage in this partition is reclaimed. During application operations, the FGS values of partitions are increased when pointers into those partitions are overwritten.

2. *Behavior.* Describes results of garbage collections. After each collection, a performance metric for the garbage collector is calculated. This metric is called the current behavior (CB). To suppress excessive noise on the behavior metric, it can be averaged over recent collections, thus introducing some form of history and deriving a history behavior (HB).

In order to combine both state and behavior to the desired metric of garbage amount, the state and behavior metrics must match. From the above design space, we derive the following heuristics.

2.4.1 Coarse Grain State / Current Behavior

To derive the amount of garbage in the system, this heuristic combines CGS with a behavior metric that is expressed as follows.

C ... bytes reclaimed in last collection

With the number of partitions (CGS) expressed as p, this results in the following equation.

$$ActGarb = C * p$$

2.4.2 Fine Grain State / History Behavior

In this heuristic, FGS is combined with a behavior metric of bytes reclaimed per pointer overwrite. We use the following definitions.

$PO(p)$ ... pointer overwrites of partition p
$GPPO$ ... garbage per pointer overwrite
$GPPO_h$ ... garbage per pointer overwrite history
h ... history factor

We use an exponential mean to derive the behavior history from current behavior according to equation

$$GPPO_h = h * GPPO_h + (1 - h) * GPPO$$

Combining state and history yields the final prediction equation.

$$ActGarb = GPPO_h * \sum_{p=1}^{n} PO(p)$$

By varying h from 1.0 to 0.0, the heuristic changes from FGS/HB to FGS/CB. The FGS/HB heuristic is very inexpensive to compute because all that is required to implement it is a single value to record the history and counters to maintain a count of the number of pointer overwrites to each partition (also necessary for the UPDATEDPOINTER partition selection policy).

3 Evaluation Method

In this section, we describe the method we use to evaluate the collection rate policies presented in the previous section. In particular, we describe the complete garbage collection algorithm into which the collection rate policies are fit, discuss the simulation techniques used in comparing the policies, and detail the test database and application used to drive the experiments.

3.1 Complete Garbage Collection Algorithm

The collection rate policies form just a part of a complete garbage collection algorithm. The partitioned collection algorithm used in our experiments is described in our previous paper [CWZ94], so we refer the reader to that paper for details. Here we give just a brief review of the important aspects of that algorithm.

We use a copying garbage collector [Che70] in which objects are relocated as a result of collection. This allows garbage collection to not only reclaim the space occupied by garbage but also to compact the collected partition's live objects for improved reference locality. Copying is done in a breadth-first traversal from the partition's roots. Copying is performed transitively from the roots until all objects are reached. Pointers leaving the collected partition are not traversed.

In our work, we decouple the issue of when to grow the database from the issue of when to collect. In particular, if an allocation is requested and there is insufficient free space anywhere in the current set of partitions, a new partition is simply added. Lack of free space never causes a garbage collection to occur, as is often done in programming language garbage collection.

We chose the I/O buffer size to be the same as the size of the partitions. We did this because a buffer significantly smaller than a partition may cause a garbage collector to perform an excessive number of I/O operations, while a much larger buffer could overwhelm any improved reference locality that resulted from the collections. In our experiments, the buffer size was set to 12 8-kilobyte pages.

3.2 Simulation Environment

Our simulation system mimics the physical and logical structure of the database implementation being measured. Traces of database application events (e.g., object creations, accesses, modifications) are used to drive the simulations; details appear in [CWZ93]. For the work described here, we use traces derived from the OO7 benchmark database [CDN93]. Details of the test database are provided in Section 3.3.

Because we are concerned with the relative performance of collection rate polices, we assume simple mechanisms for concurrency control and recovery. In particular, we make the simplifying assumptions that the entire database is locked while collection is performed, and that logging for recovery is not supported. Clearly, more sophisticated mechanisms must be provided in actual implementations; proposals for such mechanisms are discussed elsewhere [AFG95, KLW89, KW93, YNY94].

We evaluate the performance of the policies based on multiple simulation runs that differ only in the initial random number seed. In our results, we present the mean of the values over time. Each simulation run experiences a cold-start of the database. We do not, however, want to include the cold-start behavior in the calculation of means. Therefore, we isolate the preamble to the significant part of a run, keeping the preamble as short as possible by using exponentially decreasing knowledge from an oracle. Mean values are then only calculated for the significant part of simulations. Preamble lengths range from 10 to 30 collections, depending on the simulation parameters, but usually were closer to 10 than to 30. For the time-varying results shown, preambles are 10 collections.

3.3 Test Database Structure

The test database used in our measurements is the OO7 benchmark [CDN93], which was also used by Yong, Naughton, and Yu in their work on garbage collection [YNY94]. Table 1 shows how the characteristics of our Small′ OO7 database that we measured compares to the Small database used by Yong, Naughton, and Yu, and described in [CDN93]. Given these parameters,

Parameter	Small'	Small
NumAtomicPerComp	20	20
NumConnPerAtomic	3/6/9	3/6/9
DocumentSize (bytes)	2000	2000
ManualSize (kbytes)	100	100
NumCompPerModule	150	500
NumAssmPerAssm	3	3
NumAssmLevels	6	7
NumCompPerAssm	3	3
NumModules	1	1

Table 1: OO7 Benchmark Database Parameters.

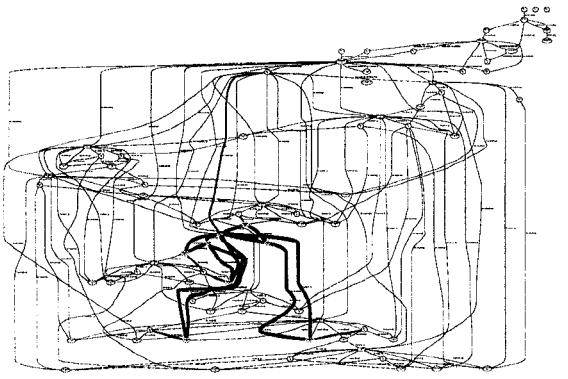


Figure 3: Example of the OO7 Database Structure.

the test database ranges from approximately 3.7 to 7.9 megabytes in size. This range allows us to run the numerous simulations required to understand the repeatability of our results. We have also experimented with applications running on a database up to 17 megabytes in size and have observed behavior consistent with the results reported in Section 4.

Figure 3 is a depiction of the structure of the OO7 database as it appears at some point during the execution of an example trace. The top level of the database is a tree hierarchy that leads to a number of composite part objects. Composite parts are composed of atomic parts and their connections, which are subordinate to the composite part. These atomic parts are highly interconnected, with an average connectivity of three. The connections highlighted in the figure, together with their associated atomic part and connection objects, form an object cluster that can be detached from the rest of the graph by overwriting just six pointers.

3.4 Test Application Behavior

Figure 2 shows the sequence of phases making up the test application. These are essentially the same phases used by Yong, Naughton, and Yu. The first phase, GenDB, generates an initial database of a particular connectivity. The second phase, Reorg1, deletes half the atomic parts and then reinserts them. The third phase, Traverse, is a read-only, depth-first traversal over all the atomic parts. Finally, the fourth phase, Reorg2, again deletes half the atomic parts and then reinserts them. Unlike the similar Reorg1, the atomic parts are reinserted in such a way as to break any clustering of atomic parts for a given composite part.

One difference from Yong, Naughton, and Yu in our use of the application is that our second and third phases are reversed. We did this in order to cause more disruption in the behavior of the application, since our goal is to test the accuracy and responsiveness to changes in application behavior. By separating the two reorganizations in our sequence, we create greater differences in phase transition. Another difference in our test application is that the original Reorg2 deleted all, rather than half, of the atomic parts. We made this change so that the two reorganizations would perform approximately the same amount of work.

4 Results

We now present the simulation results for the SAIO and SAGA collection rate policies, including the garbage estimation heuristics needed by the SAGA policy. We first investigate the effectiveness of the policies at meeting a range of user-requested settings. We then show that our collection rate policies work in databases of varying connectivities.

4.1 Accuracy and Responsiveness

In the results presented, each data point shows the mean of 10 runs, with the connectivity between atomic parts (i.e., NumConnPerAtomic) set to 3. The means shown are computed as the average sampled at each database event (i.e., object creation, access, or modification). Sampling at each event represents an approximation of a uniform sample, given the assumption of an active workload. The figures present error bars indicating the minimum and maximum means over the 10 runs. In many instances the error bars are hard to distinguish, because the range of errors is negligible.

4.1.1 SAIO Policy

Figure 4 shows the accuracy of the SAIO policy over a range of requested I/O percentages. By accuracy, in this case, we mean how well the policy is able to achieve the parameter setting *SAIO_Frac* provided by the user. Clearly, the SAIO policy is very accurate at controlling the garbage collection I/O percentage. This high accuracy comes about for three reasons. First, the control algorithm is correct, as our results show. Second, it is very effective if the input data given to it are accurate. In the case of I/O operations, the data are exact because they can be measured directly. Third, the assumptions made by the algorithm are valid. In

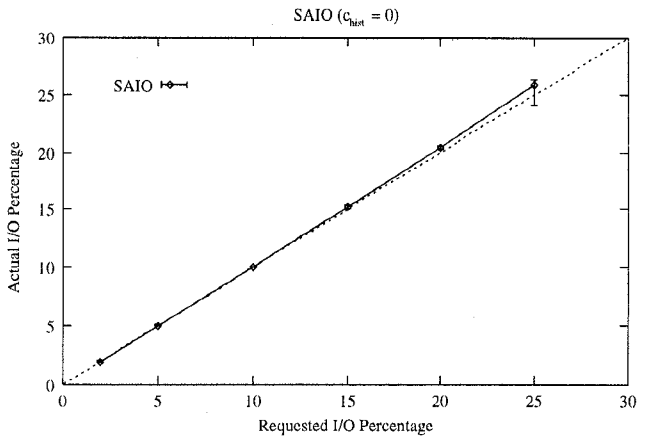


Figure 4: Effectiveness of SAIO Policy as a Function of the Requested I/O Percentage.

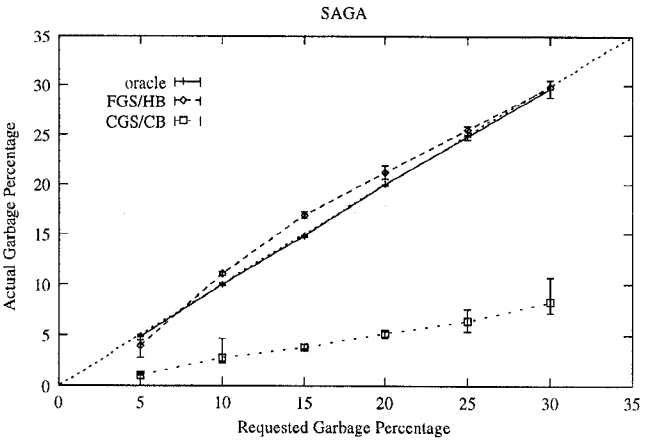


Figure 5: Effectiveness of SAGA Policy as a Function of the Requested Garbage Percentage.

particular, for the SAIO policy, we assume that the number of I/O operations from one collection to the next remains fairly constant. While this assumption breaks down occasionally (i.e., during phase changes in the application), overall it appears to hold.

The figure shows that at the highest I/O percentages, the policy results in slightly more I/O operations than requested, and there is more variance among runs. The extra I/O operations result when the primary SAIO assumption breaks down. This breakdown occurs more often at higher percentages because more collections are performed. To understand why the actual I/O percentage is higher than requested, consider the following scenario. Suppose the first, second, and third collections resulted in 100, 50, and 100 I/O operations, respectively. After the first collection, the assumption would lead us to predict that the next collection would require 100 I/O operations, which would result in a 200% error (100/50). For the third collection, the policy would predict 50 I/O operations, and the error would be 50% (50/100). As a result, the errors do not cancel ((200+50)/2=125), which causes the actual I/O percentage to drift above the requested percentage. Note that this happens only in extreme cases—in practice, such a high percentage would probably never be requested by the user.

In Figure 4 the c_{hist} parameter has been set to 0, which means that no history is used to compute the next collection interval. As a result, the policy should be highly responsive to changes in the application behavior. In fact, we simulated the entire range of I/O percentages at both extremes (i.e., $c_{hist} = 0$ and $c_{hist} = \infty$), and found that for the OO7 application the use of any amount of history makes little difference with respect to the accuracy of the policy. However, expanding the history does reduce the error seen at high I/O percentages because the error would be exposed to the control algorithm, which could then respond to eliminate it. At $c_{hist} = 0$, the control algorithm does not use misprediction information from previous collections, and thus cannot respond as accurately. While the use of history information makes little difference in the OO7 application, if the assumption that $\Delta GCIO \equiv CurrGCIO$ is violated more often in other applications, the history parameter can be used to ameliorate the impact.

4.1.2 SAGA Policy

Figure 5 shows the accuracy of the SAGA policy over a range of requested garbage percentages. Here accuracy means how well the policy is able to achieve the parameter setting *SAGA_Frac* provided by the user. Recall that the SAGA policy, unlike the SAIO policy, uses estimated information, and that these estimates come from either the CGS/CB or the FGS/HB heuristics. To evaluate the SAGA policy independent of the accuracy of the heuristics, we include results obtained using an impractical-to-implement oracle, which knows exactly how much garbage is in the database at each collection.

The figure shows that the SAGA policy using the oracle is extremely accurate, such that the line is difficult to distinguish from perfect accuracy. This result confirms two things. First, the control algorithm is correct and, second, the assumptions made in the algorithm are valid for this application. In particular, the assumptions are that successive collections collect approximately the same amount of garbage, and that the database size does not change appreciably between successive collections. The first assumption is further confirmed by results presented below.

Figure 5 also shows the results for instances of the SAGA policy using the CGS/CB and FGS/HB heuristics. As the figure shows, the CGS/CB heuristic is quite poor at achieving the requested garbage percentage, while the FGS/HB policy is much better. Note that

the error bars, especially for the FGS/HB heuristic, are very narrow. The CGS/CB heuristic shows larger error bars because the control algorithm in its case behaves much more erratically.

We now examine in detail why these heuristics differ in their accuracy and variance. In addition, we examine why the FGS/HB policy shows a systematic variation of a small amount of inaccuracy (i.e., a "bump"). Figures 6a and 6b show the time-varying behavior of CGS/CB and FGS/HB. In these figures, the requested garbage percentage is fixed at 10%. The figures show the target, actual, and estimated garbage percentage in the database as a function of the number of collections performed. The number of collections performed when using each heuristic differs because the heuristic controls the rate of collection. Note that because the SAGA policy measures time in pointer overwrites and Traverse is a read-only phase, "time" does not progress between the end of Reorg1 and the beginning of Reorg2. This makes sense because no garbage can be created during a read-only phase.

As Figure 6a clearly shows, the CGS/CB heuristic exhibits widely varying estimates of the garbage percentage, and a significant overestimation of the actual amount of garbage in the database. This behavior results directly from the nature of the heuristic, which assumes that information gained from the collection of the current partition is representative of all the partitions in the database. This assumption fails because the partition selection policy employed (i.e., UPDATED-POINTER [CWZ94]) is effective at finding a partition with more than an average amount of garbage. If the partition selection policy used was likely to find a partition with only an average amount of garbage (e.g., it picked a random partition to collect), then the CGS/CB heuristic would provide a more accurate estimate. Note also that CGS/CB uses only the current behavior (i.e., information from the current collection) to estimate the garbage percentage in the database. As a result, its garbage percentage estimates can vary dramatically from collection to collection, as the figure clearly shows.

On the other hand, FGS/HB shows a consistently accurate estimate of the percentage of garbage in the database, even when the application behavior changes (e.g., from Reorg1 to Traverse to Reorg2 at the 25th collection). This accuracy results from its use of fine grain state in the form of the number of pointer overwrites, which has been shown to be highly related to garbage creation [CWZ94]. There is also significantly less variation in the garbage estimation of the FGS/HB heuristic because of its use of historical information.

Figure 7a shows the sensitivity of the FGS/HB heuristic to changes in the history parameter, h, discussed in Section 2.4. The responsiveness of the heuristic is best understood in the context of changes in the database application behavior. The figure reflects two distinct behavior changes in the application. These changes occur at the startup of the application, when the database is generated, and at the transition between the two database reorganizations (occurring at collection number 9 in the top graph). How responsive the heuristic is to changes depends on the amount of history it uses. In the case of 95% history, we see that the heuristic is very slow at adapting to changes in the application behavior, resulting in large swings in the estimated garbage percentage and significant errors. By the 60th collection, however, the 95% history shows relatively stable and accurate estimation. On the other hand, with only 50% history, we see that the heuristic is very responsive to the application changes, but it develops systematic inaccuracies as a result. In particular, note that after 40 collections the heuristic develops an oscillation that results entirely from the mathematics of the control algorithm. The specific problem is caused by a breakdown in the SAGA policy assumption that the computed derivative, $TotGarb'(t)$, is accurate. In practice, we have used 80% history with success in this application.

Figure 7b shows different aspects of the FGS/HB heuristic as a function of the number of collections. In these graphs, the requested garbage percentage is 10% and the history parameter in FGS/HB is set to 80%. The figure shows how the collection rate, collection yield, and garbage percentage vary over time. In all of these graphs, the transition from Reorg1 to Traverse to Reorg2 occurs at the 25th collection.

Looking at the collection rate as a function of time in the top graph of Figure 7b, we see that the cold-start of the database causes initially high rates. After this initial transient, the rate settles to approximately one collection per 200 overwrites. Finally, at the Reorg1-Traverse-Reorg2 transition, the rate becomes less stable, but averages to an overall lower value.

The collection yield, shown in the middle graph of Figure 7b, indicates how the amount of garbage collected differs during different phases of the application. In this graph, there are clear differences in the collection yield caused by the two database reorganization phases. In particular, Reorg2 produces less garbage per partition as it executes. Note that the transition between reorganization phases occurs at the 25th collection, but partitions containing garbage from Reorg1 remain in the database until approximately the 35th collection. This behavior also indicates why the collection rate does not immediately decrease after the 25th collection.

The bottom graph of Figure 7b indicates the effectiveness of the garbage estimation heuristic during the different phases of the application. (This graph duplicates the middle graph of Figure 7a, but at a different scale.) Note that based on this figure alone, one might

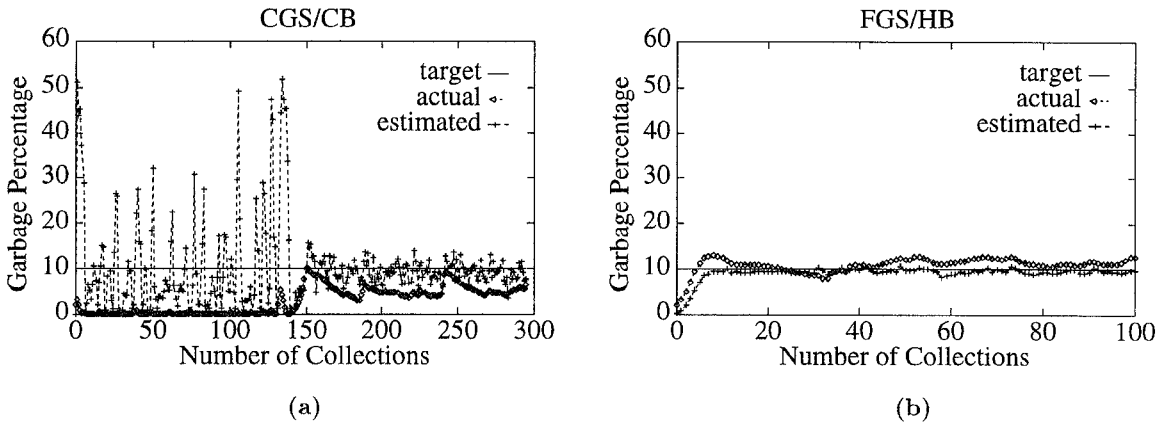


Figure 6: Time-varying Behavior of Garbage Estimation in the CGS/CB (a) and FGS/HB (b) Heuristics.

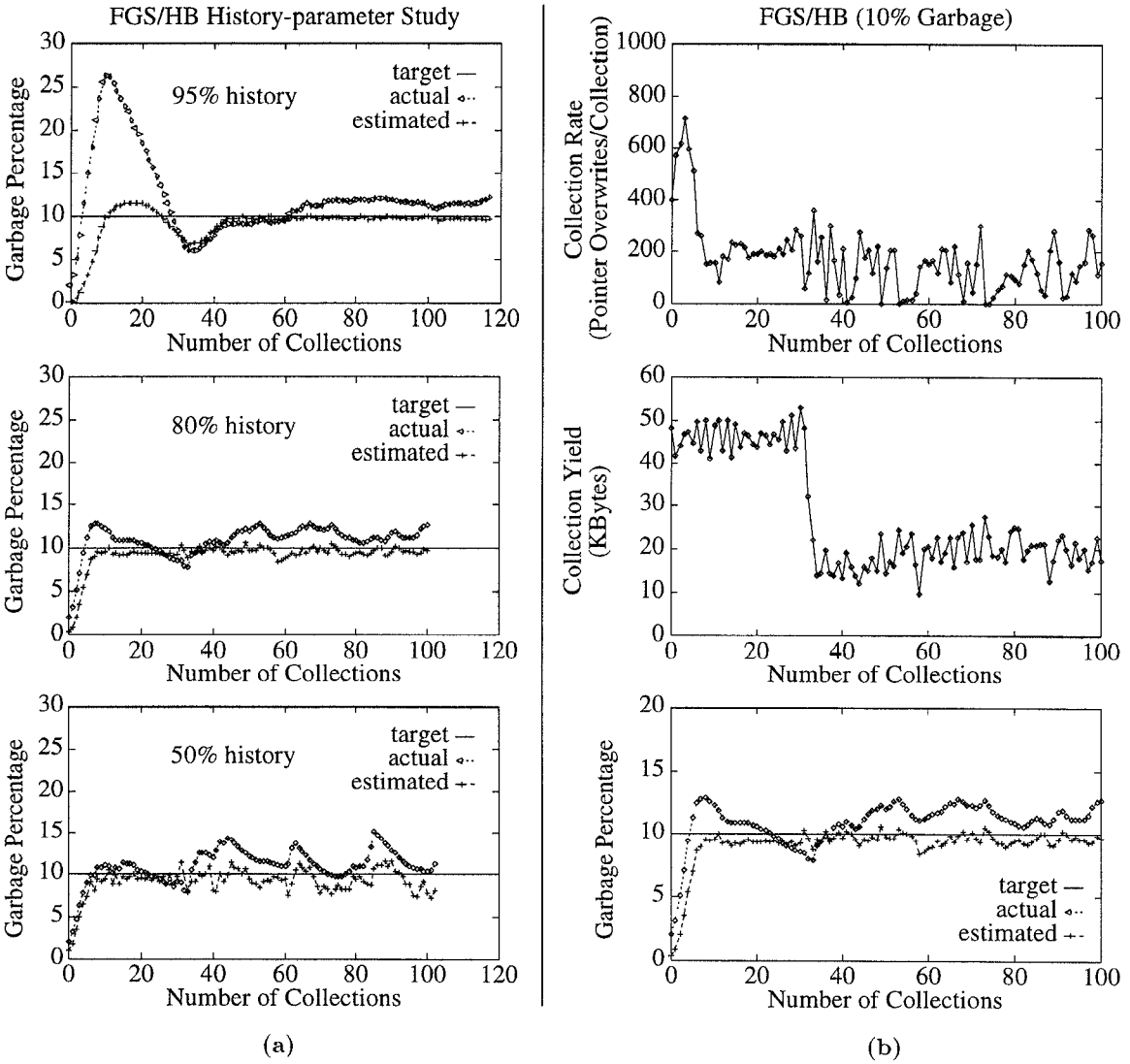


Figure 7: History Parameter Study of the FGS/HB Heuristic.

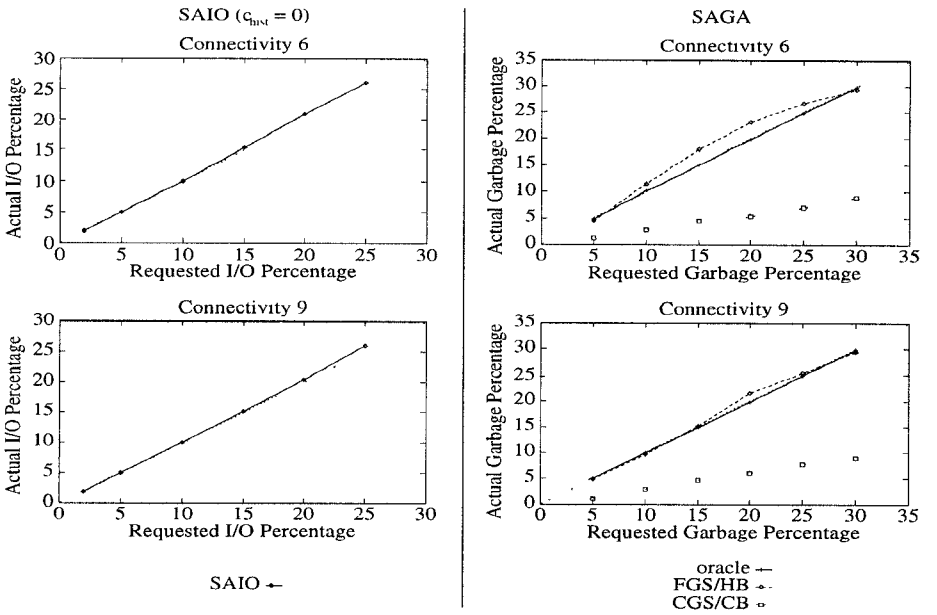


Figure 8: Sensitivity of Policy Accuracy to Database Connectivity.

conclude that the average of the actual garbage in the database was somewhat higher than the average presented in Figure 5. This observation also relates to the systematic "bump" that can be observed in Figure 5. The behavior results from the way in which the average garbage percentage is computed. As mentioned, the garbage percentage is sampled each time an application event occurs. As a result, a number of samples are included that occur during the Traverse phase (at the 25th collection). Thus, the particular garbage percentage during that period has an impact on the computed average. As the requested garbage percentages changes, there is a systematic shift in the startup curve shown in the bottom graph, resulting in a different point in the curve occurring during Traverse. This systematic error is also present, to differing degrees, when we consider databases with higher connectivity, as discussed below.

4.2 Sensitivity to Database Connectivity

Figure 8 shows the sensitivity of the SAIO and SAGA policies to changes in the connectivity of the database. In the graphs presented, each data point shows the results from one run of the OO7 database with the connectivity among atomic parts (i.e., NumConnPerAtomic) set to 6 and 9. The results in the graphs are consistent with those in figures 4 and 5, where the connectivity is set to 3. This supports the assertion that the SAIO and SAGA polices are effective across a variety of database connectivities.

5 Summary

One important aspect of garbage collection in object databases is determining how often to collect. Collecting too often results in excessive garbage collection I/O overhead and collecting too infrequently results in large amounts of garbage in the database. The proper collection rate is a function of user preferences, database structure, application behavior, and database size. Furthermore, because applications can exhibit distinctly different phases, no particular fixed collection rate can provide the desired performance. No previous work has concentrated on the problem of controlling the collection rate in object database garbage collection.

In this paper, we have proposed and evaluated two semi-automatic, self-adaptive collection rate policies. These policies are guided by user input about what level of performance is desired. In particular, the semi-automatic I/O (SAIO) policy attempts to achieve a specified level of garbage collection I/O operations as a percentage of total I/O operations, and the semi-automatic garbage (SAGA) policy attempts to achieve a specified percentage of garbage in the database. These policies are self-adaptive in that they dynamically respond to temporal changes in the application behavior.

The SAIO policy uses exact information about I/O operation counts that is typically gathered for other purposes by the ODBMS. On the other hand, the SAGA policy must use heuristics to estimate the amount of garbage in the database because determining it exactly is prohibitive. We have explored two simple and practical heuristics for this purpose.

Using the OO7 database and an application first used by Yong, Naughton, and Yu, we show that our policies are accurate at achieving a wide range of user-specified I/O and garbage percentage settings. The policies are also responsive to phase changes in the OO7

application. We also show that one of our garbage estimation heuristics, FGS/HB, was effective for this application. Our current results suggest that pursuing further investigations of these policies is worthwhile.

First, we intend to better understand the legitimacy of our assumptions. In particular, it will be important to find out whether commercial object databases and applications violate these assumptions, and if so, what impact this has on the effectiveness of the policies.

Another direction for future work is to more tightly couple the two policies with respect to achieving a global optimum. In particular, the SAIO policy could use information provided by the SAGA heuristics to determine the cost-effectiveness of the I/O operations being performed, and adjusting itself accordingly.

We also intend to place our policies in a broader context. In the current studies, we have assumed an active database workload. Our policies define a particular interval at which to do the next collection. If it appears advantageous to perform collection before the interval expires (e.g., the application workload drops to a quiescent state), then such opportunism can be considered. Semi-automatic, self-adaptive policies are well equipped to take advantage of such opportunism.

References

[AFG95] L. Amsaleg, M. Franklin, and O. Gruber. Efficient incremental garbage collection for client-server object database systems. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, September 1995.

[Bjö93] Anders Björnerstedt. *Secondary Storage Garbage Collection for Decentralized Object-Based Systems*. PhD thesis, Stockholm University, Dept. of Comp. Sys. Sciences, Royal Inst. of Tech. and Stockholm Univ., Kista, Sweden, 1993. Also appears as Systems Dev. and AI Lab. Report No. 77.

[But87] Margaret H. Butler. Storage reclamation in object-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 410–423, San Francisco, CA, 1987.

[CA86] Jack Campin and Malcolm Atkinson. A persistent store garbage collector with statistical facilities. Persistent Programming Reserarch Report 29, Department of Computing Science, University of Glasgow, Glasgow, Scotland, 1986.

[Cat93] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1993.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 12–21, Washington, DC, June 1993.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Comm. of the ACM*, 13(11):677–678, November 1970.

[Cor94] Servio Corporation. Announcing GemStone version 4.0. Product literature, 1994.

[CWZ93] Jonathan Cook, Alexander Wolf, and Benjamin Zorn. The design of a simulation system for persistent object storage management. Technical Report CU-CS-647-93, Department of Computer Science, University of Colorado, Boulder, CO, March 1993.

[CWZ94] Jonathan Cook, Alexander Wolf, and Benjamin Zorn. Partition selection policies in object database garbage collection. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 371–382, Minneapolis, MN, March 1994.

[KLW89] Elliot Kolodner, Barbara Liskov, and William Weihl. Atomic garbage collection: Managing a stable heap. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 15–25, Portland, OR, June 1989.

[KW93] Elliot Kolodner and William Weihl. Atomic incremental garbage collection and recovery for a large stable heap. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 177–186, Washington, DC, June 1993.

[Mat85] David C. J. Matthews. Poly manual. *SIGPLAN Notices*, 20(9), September 1985.

[ML94] Umesh Maheshwari and Barbara Liskov. Fault-tolerant distributed garbage collection in a client-server, object-oriented database. In *Proceedings of the Parallel and Distributed Information Systems*, pages 239–248, Austin, TX, September 1994.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, St. Malo, France, September 1992.

[WJMK94] Jr. William J. McIver and Roger King. Self-adaptive, on-line reclustering of complex object data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 407–418, Minneapolis, MN, March 1994.

[YNY94] Voon-Fee Yong, Jeffrey Naughton, and Jie-Bing Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Proc. of the 10th International Conference on Data Engineering*, pages 120–131, February 1994.