Effectively Controlling Garbage Collection Rates in Object Databases

Jonathan E. Cook†, Artur W. Klauser‡, Alexander L. Wolf†, and Benjamin G. Zorn†

†Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

‡Institute for Applied Information Processing
Graz University of Technology
8010 Graz, Austria

University of Colorado
Department of Computer Science
Technical Report CU-CS-758-94 October 1994

October 1994

Abstract

A fundamental problem in automating object database storage reclamation is determining when and 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 range of test databases. The evaluation demonstrates that semi-automatic, self-adaptive policies are a practical means for flexibly controlling garbage collection rate without sacrificing overall application performance.

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 [9, 13, 14, 20]. A recently proposed standard suggests using garbage collection for at least some of the programmatic interfaces to an ODBMS [5]. Commercial ODBMSs are even beginning to provide implementations of garbage collection (e.g., [10]).

In a previous paper [9] 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 [20] and is akin to generational collection in programming language systems [19]. We categorized the issues into a number of policy areas that together contribute to a complete garbage collection algorithm. We described our results in investigating one policy area, *partition selection*, which is the selection of which partition of a database to collect during a given garbage collection. We introduced a new partition selection policy, called UPDATEDPOINTER, and showed that it performed better than all other existing policies and close to a near-optimal, but unimplementable, selection policy over a wide range of database sizes and object connectivities.

In this paper we investigate another critical policy area of partitioned garbage collection algorithms, that of determining when and 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 1 and 2 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 [4], are discussed in Section 4.3.) The figures highlight the time/space tradeoff

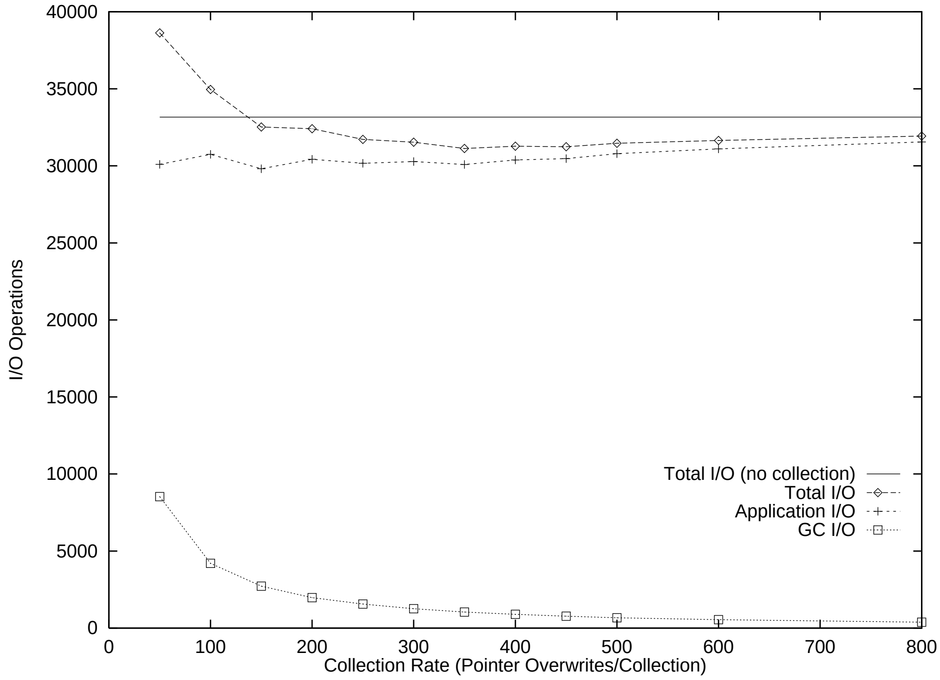

Figure 1: Collection Rate versus Application and GC I/O Operations.

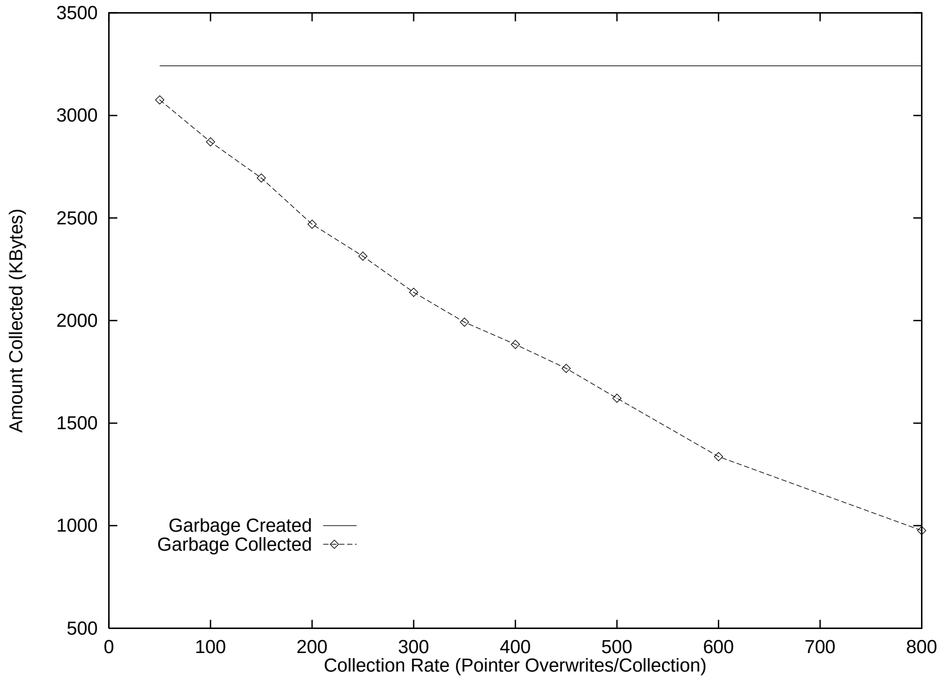

Figure 2: Collection Rate versus Total Garbage Collected.

of collection rate policies. For example, it is clear that choosing a collection rate of 50, measured in pointer overwrites (i.e., modifications of pointers between objects) per collection, results in excessive numbers of I/O operations, while choosing a collection rate of 800 pointer overwrites per collection results in little garbage being collected. 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 memory overhead. This issue is best decided by the ODBMS user,[1] 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 figures 1 and 2. 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 both *semi-automatic*, in that the ODBMS, rather than the ODBMS implementor or the ODBMS user, should set the rate, and *self-adaptive*, in that the collection rate can vary in response to the dynamic behavior of the applications manipulating the database. The role of the ODBMS implementor and user should be to provide the broad performance goals that implicitly guide the control of the collection rate by the ODBMS itself.

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 responsive 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

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

percentage. As the mix of I/O operations changes, the collection rate adjusts to maintain the target percentage.

We present a performance evaluation of our two new collection rate policies based on trace-driven simulations of two, very different database applications. One is the application developed by Yong, Naughton, and Yu [20] for the OO7 benchmark [4] and the other is based on the our own augmented binary tree application [9]. Our results show that our semi-automatic collection rate policies give excellent performance in all cases and often provide performance close to that of the best fixed-rate policy. By "best", we mean the fixed rate that corresponds exactly to the user-specified constraint. Furthermore, our collection rate policies are simple to compute and require virtually no additional memory. We also show that our results scale to databases of different sizes.

The paper has the following organization. Section 2 provides background material and describes related work. Section 3 describes our new collection rate policies in more detail. Section 4 describes our experimental methods used to evaluate these policies, and the test databases used in the evaluation. Section 5 presents our results and Section 6 summarizes our findings.

2 Related Work

The related work in this area falls into three distinct categories: collection rate determination in programming language garbage collection, other research in object database garbage collection, and research in on-line database reclustering.

2.1 Programming Language GC Algorithms

Primary memory garbage collection algorithms also must determine the most appropriate collection rate. One approach often taken by these algorithms is to collected "as needed". For example, many algorithms will invoke collection whenever the free-list (in non-copying algorithms), or current semispace (in copying algorithms) is exhausted. This approach has the disadvantage that if little garbage is reclaimed during a collection, then the next collection may occur very shortly thereafter, after the little storage that was reclaimed is exhausted. Usually such policies include heuristics for growing the heap if not enough storage is reclaimed by a collection [7, 19].

Another common approach to determining when to collect is to collect after a fixed amount of storage has been allocated. For example, in some systems, collections will occur after every megabyte of storage is allocated. These schemes avoid the potential thrashing behavior associated with the "collect-as-needed" policies. Furthermore, they are motivated by the empirical observation that most allocated programming language objects become garbage very soon after they are allocated [17, 21]. Some collector implementations allow the user to specify the fixed amount of allocation (i.e., specify collection after every two megabytes of allocation) to tune the collection rate to the behavior of the program [11].

There are many differences between programming language and object database garbage collection with respect to collection rate. One important objective of programming language collection is to collect frequently enough that the program and its data fits well in the primary memory. Thus, collections are usually triggered after at most a few megabytes of allocation. It is not clear what the analog to this objective is in object database garbage collection. In addition, programming language collectors are limited in what information they can use in making policy decisions. In particular, whereas there is relatively low overhead in counting the number of pointer stores performed in an object database, counting the stores in a programming language algorithm would incur significant overhead.

2.2 Object Database Garbage Collection

There has been a significant amount of research in object database garbage collection, much of which is quite recent [1, 2, 3, 9, 16, 13, 14, 20]. Here, we discuss the research that most closely relates to our current work. Both Butler [2] and Yong et al [20] show that existing programming language garbage collection algorithms have inadequate performance when used in database systems. Yong et al. [20] propose a technique called "partitioned garbage collection" for object databases that represents a generalization of the programming language garbage collection technique called generation garbage collection [15]. With partitioned garbage collection, a subset of the entire database is collected incrementally and independently of the rest. Collection with their algorithm is triggered either when free-space becomes unavailable or after a fixed amount of storage is allocated; these techniques are both taken directly from programming language GC algorithms.

In previous work, we investigated the impact of the partition selection policy on the performance of partitioned garbage collection in object databases [9]. In that work, we assumed that the collection rate was set at a fixed value. This paper compliments our previous work by investigating another, orthogonal garbage collection policy, that of determining the collection rate. In this work, we build on the results of our previous work by using the most effective partition selection policy previously discovered. We also extend the domain of our simulations in this work by including results from the OO7 benchmark [4]. In this paper we again assume that a partitioned garbage collection algorithm is used. For a detailed discussion of the implementation issues related to partitioned garbage collection, we refer readers to our previous paper.

None of the previous work on object database garbage collection has investigated the issue of collection rate specifically. Our work both quantifies the cost of poor collection rate choices and proposes new policies for effectively controlling the collection rate.

2.3 On-line Reclustering Algorithms

There are a number of similarities between the process of on-line reclustering of objects for performance reasons and that of garbage collection. In particular, they both have the effect that by relocating objects they have the potential to improve the application performance. Similarly, they both incur additional execution overhead that must be balanced against the performance benefits they provide. Thus, controlling the rate of both on-line reclustering and garbage collection is very important.

In recent work, McIver and King investigate the performance of on-line reclustering in object databases[18]. In their work, reclustering is triggered when a measure of reference locality (the "external tension"), normalized to the number of disk accesses, exceeds a certain threshold and when an cluster analysis module determines that reclustering should improve performance. Because on-line reclustering does not attempt to reduce the size of the database, the overall goal of this work was to reduce total application I/O operations. Our work differs from theirs because with garbage collection, the added dimension of database growth due to uncollected garbage must be considered.

3 Collection Rate Policies

Given that the goal of a garbage collection algorithm is to collect when we can reclaim the most garbage, it is natural to consider what identifiable events occur that will indicate when garbage has been created. A heuristic used in programming language GC algorithms is that new object allocation and garbage creation are correlated, thus collecting after allocating a fixed number of bytes is motivated. However, allocation and garbage creation are not always correlated. On the other hand, we know that when pointers are overwritten, the objects in the database 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 to heuristically provide a signal that garbage is being created in the database. We have used this metric in our previous work as well.

Having motivated our decision to use pointer overwrites as a basis for determining when to collect, we now discuss alternative policies for controlling the collection rate.

3.1 Fixed Rate Policies

The collection rate policy that uses the least information is a policy that fixes the rate of collection over all applications. For example, the GC algorithm implementor might decide that garbage collection should be invoked after every 500 pointer overwrites in the database. The most important question associated with this policy is how does one choose the fixed rate? Clearly any particular choice will not be optimal for all applications. Furthermore, from figures 1 and 2 we can see that selecting too high a rate can lead to many GC I/O operations. On the other hand, being too conservative about the rate is likely to lead to large quantities of garbage in the database. Therefore, any fixed-rate policy that ignores characteristics of the application or collector implementation is likely to fail.

On the other hand, a clever fixed-rate policy could attempt to determine the collection rate based on characteristics of the application, such as connectivity and object size, and characteristics of the ODBMS, such as partition size. For example, we know that our OO7 application has an approximate average connectivity of four (i.e., each object has 4 pointers pointing to it), and that objects are 133 bytes on average. From this, we could infer that every four pointer overwrites creates 133 bytes of garbage. Since our partitions are 96 kilobytes in size, then an obvious choice for collection rate would be to collect every 2956 pointer overwrites (i.e., when a partion's worth of garbage has been created). Unfortunately, this simple heuristic also fails miserably. The OO7 application mentioned actually creates garbage at a rate of 1 kilobyte per 6 pointer overwrites, or five times more garbage than the simple calculation would predict.

There are two reasons 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 fail to adapt to changes in the database behavior. The OO7 applications, 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 both reorganizations. Finally, even if a particular fixed rate is known to give specific performance for a database of one size, there is no obvious way to scale the rate for larger database sizes (as we show in Section 5). Thus, a particular fixed rate is not only application specific, but also database size specific. Clearly fixed-rate policies are unacceptable.

3.2 Semi-Automatic, Self-Adaptive Rate Policies

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 input. The first policy attempts to limit the I/O operations associated with garbage collection 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. In this section, we describe how these policies work and how they should be implemented. Furthermore, both of these policies are self-adaptive, that is, they adjust the collection rate dynamically as the database application behavior changes.

3.2.1 A Collection Rate Policy Based on I/O Percentage

With the SAIO policy, the database user indicates what fraction of application I/O operations should be used to perform garbage collection. For example, if the user wanted garbage collection to use approximately 10% of the total application I/O operations, the user would set the SAIO value (called $SAIO\_Fraction$ below) to 10%. The policy we describe uses information about the current number of GC and application I/O operations and the number of I/O operations expected during the next garbage collection to determine when the next garbage collection should occur. Note that since we are linking the collection rate to the number of pointer overwrites, in all the equations we present, time is measured in pointer overwrites. This metric makes sense because if no pointers are being overwritten, then by definition, no more garbage is being created.

First, consider the following definitions:

$$\begin{aligned} TotalIO(t) &= \text{Total application I/O operations by time } t \\ GCIO(t) &= \text{Total garbage collection I/O operations by time } t \\ TargetGCIO(t) &= \text{Target GC I/O operations by time } t \\ &= TotalIO(t) * SAIO\_Fraction \\ IODiff(t) &= GCIO(t) - TargetGCIO(t) \\ CurrentGCIO &= \text{number of I/O operations caused by the current collection} \end{aligned}$$

Stated in these terms, the goal of the SAIO policy is to make $GCIO(t) = TargetGCIO(t)$, or $IODiff\ (t) = 0$. With the above definition, if $IODiff\ (t) > 0$, we have done too many GC I/O operations, and if $IODiff\,(t) < 0$, we have done too few GC I/O operations.

Now consider that we have just completed a garbage collection at time t that resulted in $CurrentGCIO$ I/O operations. The goal of the SAIO policy is the determine at what time $t + \Delta t$ to schedule the next garbage collection. As a simplifying assumption, we further estimate that the next collection will result in the same number ($CurrentGCIO$) of I/O operations as the current collection. Then the following relations hold:

$$\begin{aligned} GCIO(t+\Delta t) &= GCIO(t) + CurrentGCIO \\ TargetGCIO(t+\Delta t) &= TargetGCIO(t) + TargetGCIO'(t) * \Delta t \end{aligned}$$

where $TargetGCIO'(t)$ is the slope of the $TargetGCIO(t)$ function. Recall that our goal is:

$$GCIO(t+\Delta t) = TargetGCIO(t+\Delta t)$$

Setting the two sides equal we get:

$$GCIO(t) + CurrentGCIO = TargetGCIO(t) + TargetGCIO'(t) * \Delta t$$

Another way to understand this last equation is the following. At some time in the future $(t + \Delta t)$, another collection will take place. At that time, we expect $CurrentGCIO$ more GC I/O operations, and $TargetGCIO'(t) * \Delta t$ more "target" GC I/O operations. The goal of the SAIO policy is to find a Δt for which this relation holds. Simplifying and solving for Δt, we get:

$$\begin{aligned} CurrentGCIO &= (TargetGCIO(t) - GCIO(t)) + TargetGCIO'(t) * \Delta t \\ CurrentGCIO &= TargetGCIO'(t) * \Delta t - IODiff(t) \end{aligned}$$

$$\Delta t = (IODiff(t) + CurrentGCIO)/TargetGCIO'(t) \quad (1)$$

To understand the impact of $IODiff(t)$ on Δt, we see that if $IODiff(t) > 0$, then Δt increases, meaning we decrease the rate of collection, and if $IODiff(t) < 0$, then Δt decreases, meaning we increase the rate of collection.

To implement this policy, the collector must be able to determine the total number of application and collection I/O operations that have been performed at each garbage collection ($TotalIO(t)$ and $GCIO(t)$) and the number of I/O operations performed during the current collection ($CurrentGCIO$). Fortunately, this information is already typically available. The implementation must also estimate $TargetGCIO'(t)$, which is equivalent to $TotalIO'(t) * SAIO\_Fraction$. Thus, the collector must maintain some history information about the total application I/O operations performed to estimate how many will be performed in the future. In the simulated implementation of this policy, we estimate $TotalIO'(t)$ using a simple formula. Given a previous slope estimate, $TotalIO'(t_{prev})$, a previous pair of data points $(t_{prev}, TotalIO(t_{prev}))$, and a current set of points $(t, TotalIO(t))$, we estimate:

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

where $Weight$ is a controlling factor that buffers the policy from rapid changes in slope. We currently set $Weight = 0.70$. Also note that in practice, Δt can become very large, if $TargetGCIO'(t)$ approaches zero, or even negative, if $IODiff(t)$ becomes too small. As a result, we place a minimum and maximum on the value of Δt. For the minimum, we use $\Delta t_{min} = 2$ and for the maximum, we use $\Delta t_{max} = 1000$. In Section 1, we have shown that a fixed collection rate of 50 results in many collection I/O operations, while a collection rate of 800 results in large quantities of garbage. Using a value of $\Delta t_{min} = 2$ guarantees an almost immediate collection, while a value of $\Delta t_{max} = 1000$ guarantees that collection occurs often enough that the policy continues to be provided with data. We find that our policy works well in practice and that Δt_{min} and Δt_{max} are rarely utilized by the policies.

3.2.2 A Collection Rate Policy Based on Percentage of Garbage

With the SAGA policy, the database user indicates what fraction of the database should contain garbage. For example, if the user wanted garbage to account for approximately 20% of the total database size, the user would set the SAGA value (called $SAGA\_Fraction$ below) to 20%. Not surprisingly, the policy is very similar to the SAIO policy and we present the most important equations in this section. First, we give the definitions:

$$
\begin{aligned}
DatabaseSize(t) &= \text{total database size by time } t \\
TotalGarbage(t) &= \text{total garbage generated by time } t \\
TotalCollected(t) &= \text{total garbage collected by time } t \\
ActualGarbage(t) &= \text{actual database garbage by time } t \\
&= TotalGarbage(t) - TotalCollected(t) \\
TargetGarbage(t) &= \text{target database garbage by time } t \\
&= DatabaseSize(t) * SAGA\_Fraction \\
GarbageDiff(t) &= ActualGarbage(t) - TargetGarbage(t) \\
CurrentCollected &= \text{amount of garbage collected by the current collection}
\end{aligned}
$$

At time t, assume we have collected $CurrentCollected$ garbage. Further, assume we will collect the same amount of garbage at the next collection. Another assumption we make is that the database size will not grow significantly between t and $t + \Delta t$. Thus, $TargetGarbage(t) \approx TargetGarbage(t + \Delta t)$.

$$
\begin{aligned}
ActualGarbage(t + \Delta t) &= TotalGarbage(t + \Delta t) - TotalCollected(t + \Delta t) \\
&= TotalGarbage(t + \Delta t) - (TotalCollected(t) + CurrentCollected) \\
TotalGarbage(t + \Delta t) &= TotalGarbage(t) + TotalGarbage'(t) * \Delta t
\end{aligned}
$$

$$\begin{aligned} ActualGarbage(t + \Delta t) &= TotalGarbage(t) + TotalGarbage'(t) * \Delta t \\ &\quad -(TotalCollected(t) + CurrentCollected) \\ &= ActualGarbage(t) + (TotalGarbage'(t) * \Delta t) - CurrentCollected \end{aligned}$$

With the assumption of insignificant database growth, our goal is:

$$ActualGarbage(t + \Delta t) = TargetGarbage(t + \Delta t) \approx TargetGarbage(t)$$

Simplifying and solving for Δt, we get:

$$\begin{aligned} TargetGarbage(t) &= ActualGarbage(t) + (TotalGarbage'(t) * \Delta t) - CurrentCollected \\ CurrentCollected &= (ActualGarbage(t) - TargetGarbage(t)) + (TotalGarbage'(t) * \Delta t) \\ CurrentCollected &= GarbageDiff(t) + (TotalGarbage'(t) * \Delta t) \end{aligned}$$

$$\Delta t = (CurrentCollected - GarbageDiff(t))/TotalGarbage'(t) \tag{2}$$

As with the SAIO policy, we place lower and upper bounds on Δt. We also approximate $TotalGarbage'(t)$ using the same formula used to approximate $TotalIO'(t)$. The major difference between the SAIO policy and the SAGA policy is that the information needed to compute formula (2) is not available exactly. In particular, $ActualGarabage(t)$ cannot be determined without scanning the entire database. As a result, to implement the SAGA policy, heuristics must be employed that estimate the current amount of garbage in the database. We have investigated three such heuristics and also implemented in our simulator an garbage estimation heuristic that actually does know how much garbage exists in the database. Our goal is to determine how effective our heuristics are relative to the garbage estimation oracle. Below, we describe how the heuristics are computed.

The SIMPLE heuristic is the easiest to compute and the most conservative. After a garbage collection has been performed, it estimates the current amount of garbage in the database by multiplying the garbage reclaimed from the current partition (the value $CurrGarbage$ from the equations above) by the number of partitions. $CurrGarbage$ can be easily determined by noting how much free space is available in the partition before and after it is collected.

The GLOBAL heuristic estimates garbage by relating the amount of garbage reclaimed to the number of overwritten pointers into a partition. Recall that in previous work [9], we showed that using the number of overwritten pointers into a partition was an excellent heuristic for identifying the partition with the most garbage. We extend that notion with this heuristic in the following way. Each time the collection of a partition occurs, we record how many overwritten pointers into that partition existed and we also record how much garbage was reclaimed when the partition was collected. This relation gives us an estimate of the amount of garbage that exists per overwritten pointer in the database. The GLOBAL heuristic computes the average garbage per overwritten pointer as a single value that is recomputed each time a partition is collected. To estimate the current amount of garbage in the database, we just multiply the current number of overwritten pointers (maintained as a result of using the UPDATEDPOINTER partition selection policy) by the global garbage per overwritten pointer value.

The LOCAL heuristic is a refinement of the global heuristic. With the LOCAL heuristic, we maintain the average garbage per overwritten pointer on a per-partition basis. The intuition is that the relation between garbage and overwritten pointers may be dependent on the partition. To compute the total garbage in the database, we sum over all the partitions the product of the number of overwritten pointers into that partition times the average garbage per overwritten pointer computed for that partition. In the cases where partitions have not yet been collected, and thus do not have an individual collection history, we use the average as computed by GLOBAL instead.

The GARBAGEORACLE heuristic is an impractical-to-implement policy that actually knows how much garbage currently exists in the database. We implement the GARBAGEORACLE heuristic in our simulator and compare the performance of the other heuristics to it.

4 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 databases and applications used to drive the experiments.

4.1 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 the same as the one described in our previous paper [9], 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 [6] 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, which varied depending on the size of the simulation run. 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 sizes range from 12 8-kilobyte pages for the smaller database simulations to 96 8-kilobyte pages for the largest database simulations.

In terms of selecting a partition from which to reclaim garbage, all the test databases use the UPDATEDPOINTER policy, which we previously showed to be superior to other existing policies [9].

4.2 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 [8]. For the work described here, we use traces derived from two sources: our own synthetic database, ABT, which we have used in previous work, and the OO7 benchmark database [4]. Details of the test databases are provided below.

One advantage of using trace-driven simulation is that we are able to evaluate and compare the performance of impractical-to-implement heuristics like the SAGA GARBAGEORACLE policy. As a result, we are able to determine how effective our heuristics are when compared to having perfect knowledge. Another advantage of using trace-driven simulation based on a synthetic database is that we are able to investigate the performance of the policies over a broad range of simulation parameters, including the database connectivity, database size, object size, partition size, large-object frequency, and the like. With this flexibility, we can establish a clear understanding of the sensitivity of our policies to the values of these parameters. On the other hand, gathering and using traces from a benchmark database, like OO7, gives us confidence that our results extend beyond this synthetic database alone.

The cost model used to evaluate the performance of the simulated algorithms is based on tracking the number of page I/O operations over the life of the simulation. We model a single-process application sharing a buffer with the ODBMS, which executes on the same processor. We simulate the database's I/O buffer and determine the number of disk I/O operations needed for each read and write. To determine when disk I/O operations take place, we simulate a database I/O buffer of a particular size (a parameter to the simulation), using an LRU policy for page replacement and a write-back scheme for updating pages. More detailed cost models can be built that would derive actual disk costs in terms of head seek, rotational delay, and transfer times, or that might model network costs for a distributed or client/server database.

Whenever possible, 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 and standard deviation of the values obtained. This includes a measure of the "average" amount of garbage in the database. An important question with respect to this measure is how to compute the average. If we sample the average at every event and compute starting from the first event, then cold-start behavior will incorrectly be included. In particular, many samples where no garbage exists at all will be recorded as the database is constructed. As a result, we compute average as follows. We first compute an incorrect average including cold-start behavior, starting the averaging at the time that garbage is first created (call it $G_{invalid}$). We then use $G_{invalid}$ to tell us when to really start computing the average by identifying at what time the fraction of garbage in the database is 75% of $G_{invalid}$. We then compute the average amount of garbage in the database by sampling from that time forward.

4.3 Test Databases

To gain an understanding of the performance characteristics of the collection rate policies under a variety of conditions, we test them using two, quite different kinds of databases. They differ mainly in their degrees of object connectivity and in the sorts of applications that are run against them.

The first test database, which we refer to as ABT (Augmented Binary Trees), consists of objects structured as a forest of binary trees, where each binary tree is augmented with some number of non-tree edges. The non-tree edges connect random nodes in the same tree and are created by a random process, but we always keep the connectivity rather low (less than 1.2). The size of objects in the database is randomly distributed around an average of 100 bytes. We do, however, include a few large leaf objects that average 64 kilobytes each and comprise about 20% of the space of all objects. We use synthetic, probabilistic models of application behavior to generate specific traces of object creations, accesses, and modifications for use in experiments. The ratio of edge reads to edge writes is not explicitly specified but rather arise from the probabilities. In our experiments, the edge read/write ratio varies from about 15 to 20. Garbage is generated by randomly overwriting tree edges from the binary trees. All, part, or none of the subtree that the overwritten tree edge pointed to may become garbage, however, because of the presence of the non-tree edges. Further details of ABT, its applications, and justifications for its usefulness are given elsewhere [9].

Figure 3 is a depiction of the structure of an ABT database as it appears at some point during the execution of an example trace. The figure was generated automatically from that trace and drawn using DOT [12].

The second test database is the OO7 benchmark, which was also used by Yong, Naughton, and Yu in their work on garbage collection [20]. OO7 has been extensively described elsewhere [4]. For our purposes, it is important to note that OO7 has a much higher connectivity than ABT and that OO7 applications, unlike ABT applications, operate in distinct phases of long object traversals intersperses with significant database reorganizations. Figure 4 is a depiction of the structure of an OO7 database, also generated from an example trace and drawn using DOT. The figure shows the thread of the top level tree hierarchy which leads to one composite part object (top-right part),

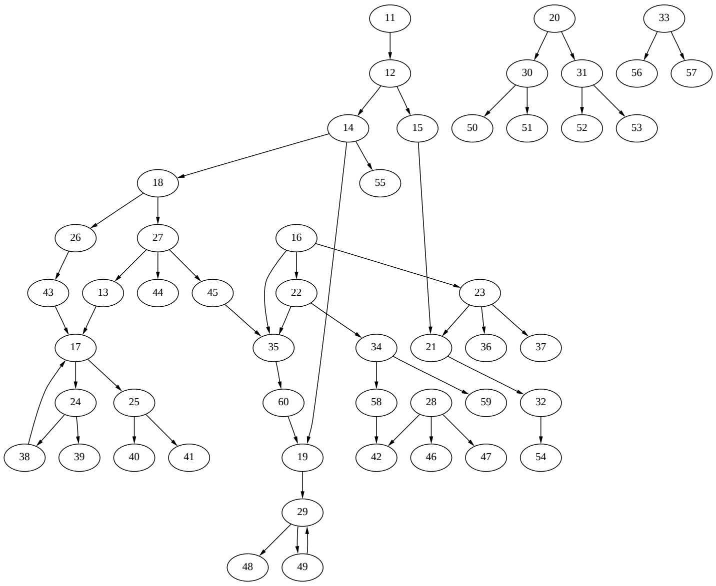

Figure 3: Example of the ABT Database Structure.

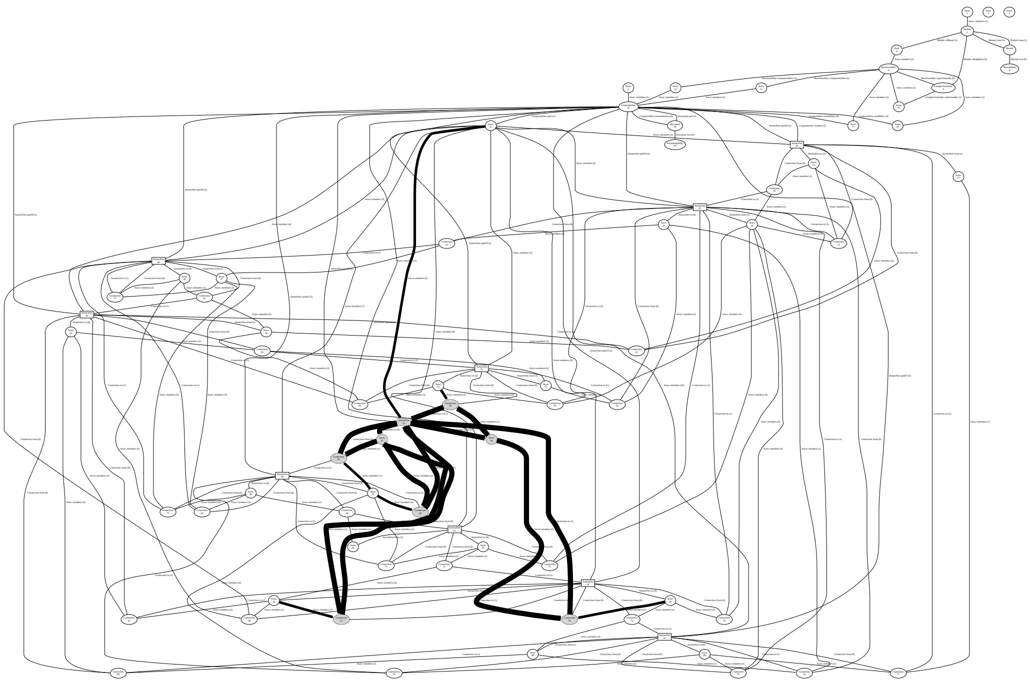

Figure 4: Example of the OO7 Database Structure.

with the rest of the figure showing the atomic parts and their connections, which are subordinate to this composite part. These atomic parts are highly interconnected, with a average connectivity of three in this case. The highlighted connections and their associated atomic part and connection objects form an object cluster that can be detached from the rest of the graph by overwriting only six pointers.

5 Results

In this section, we present results of measuring the collection rate policies we have described and also the garbage estimation heuristics needed by the SAGA policy. We have several goals in this section. First, we investigate the effectiveness of the semi-automatic policies in achieving the parameter settings the user specified. Next, we investigate the overall performance of our collection rate policies and compare them against fixed-rate policies in which the user was able to correctly guess what fixed-rate value to use to achieve the desired performance. Finally, we show that our collection rate policies work in databases of varying sizes.

5.1 Accuracy of the Semi-Automatic Collection Rate Policies

In Section 3, we describe two semi-automatic collectioned rate polices: SAIO, which attempts to control the I/O operations required during garbage collection, and SAGA, which attempts to control the amount of garbage in the database. In this section, we show how accurate these policies were at achieving the parameter setting specified by the user. In particular, we investigate the impact of the simplifying assumptions made and also the use of heuristics to estimate the garbage in the database.

Figure 5 shows the accuracy of the SAIO heuristic in both the ABT and OO7 database applications. The figure shows the relation of the requested value of $SAIO\_Fraction$ to the actual fraction of garbage collection I/O operations that resulted from using the heuristic. We see that in the ABT application, the SAIO heuristic comes very close to the performance specified by the database user. On the other hand, in the OO7 application, the SAIO heuristic clearly undershoots the user-specified value of $SAIO\_Fraction$. To understand why the SAIO policy undershoots the requested value, consider Figure 6. This figure shows the fraction of garbage collection I/O opera-

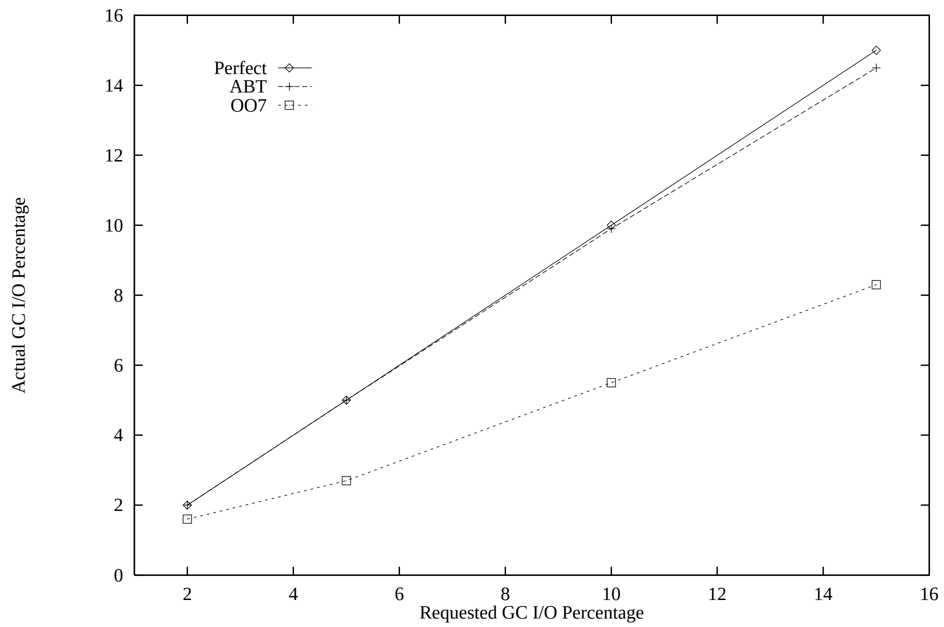

Figure 5: Effectiveness of SAIO Heuristics as a Function of Percentage I/O Operations Allowed in the ABT and OO7 Database Applications.

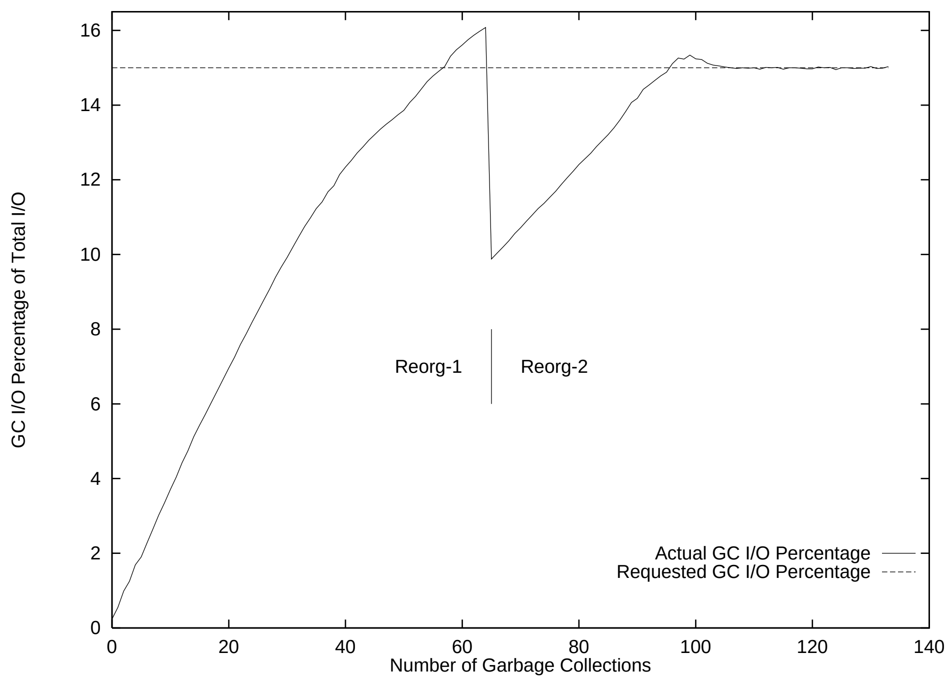

Figure 6: Effectiveness of the SAIO Policy in the OO7 Application as a Function of Time.

tions performed as a function of time in the OO7 application with *SAIO_Fraction* set to 15%. The data in the figure were recorded every time garbage collection was performed. The figure shows that as time increases, the GC I/O percentage increases to the requested value, however it does so at a relatively slow rate. Also, the OO7 application has two distinct reorganization phases separated by traversals in which no pointer overwrites occur. As a result, the GC I/O percentage drops precipitously after approximately 60 collections, which occurs because there is a long period in which only application I/O operations are generated. From the figure, we see that while the average GC I/O percentage is well below the requested 15%, the policy achieves a GC I/O percentage very close to the requested value and the low average is primarily caused by cold-start effects. Thus, we conclude that the SAIO policy is very effective at achieving the requested GC I/O percentage in both databases, assuming the application executes long enough and performs pointer overwrites on a regular basis as it executes.

Figures 7 and 8 show the performance of the SAGA collection rate policy using the garbage estimation heuristics described in Section 3. In these experiments, the requested garbage percentage was varied from 10% to 30%. First, we see that in both databases, the GarbageOracle heuristic (which knows exactly how much garbage is in the database) performs very well. Thus, the simplifying assumptions we made are supported. Next, we see that the Simple heuristic performs well in the ABT database but poorly in the OO7 database. This result is explained by the pattern of garbage creation in the two databases. In the ABT database, garbage is generated at random throughout the database, while in the OO7 database, during the first reorganization, garbage is generated very locally. Thus, the Simple heuristic significantly overestimates the amount of garbage in the database (based on its assumption that all partitions contain as much garbage as the partition currently being collected). In this case it collects too frequently, which results in the actual garbage percentage being much less than the requested percentage. Simple is the most appropriate heuristic if a conservative garbage estimate is desired (thus allowing too little garbage in the database is more acceptable than allowing too much), or if the database structure is known to be relatively homogeneous (i.e., garbage is likely to be spread evenly throughout the database).

The Global and Local garbage estimation heuristics appear to be the best estimators when considering their performance in both databases. In the ABT database, these two heuristics tend to underestimate the amount of garbage (leading to more garbage in the database then requested),

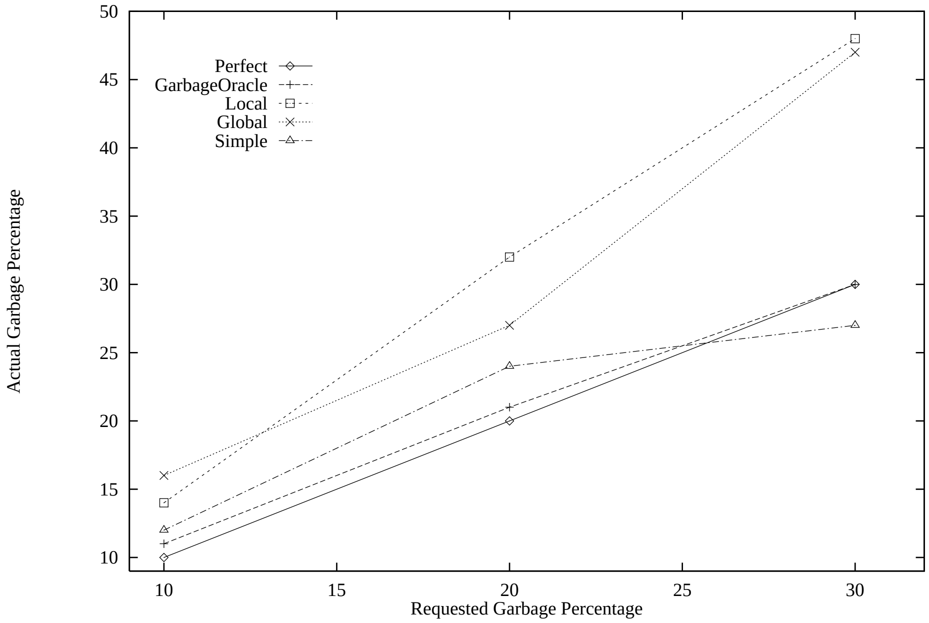

Figure 7: Effectiveness of SAGA Heuristics as a Function of Percentage Garbage in the ABT Application.

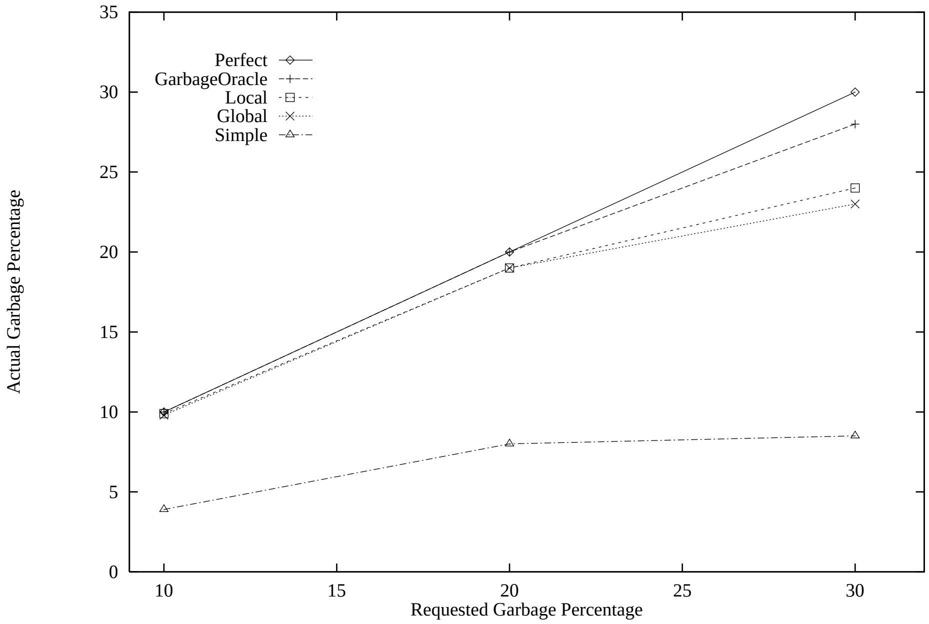

Figure 8: Effectiveness of SAGA Heuristics as a Function of Percentage Garbage in the OO7 Application.

whereas in the OO7 database these heuristics are close to optimal. While the two heuristics are quite close to each other in accuracy, the GLOBAL heuristic performs slightly better than the LOCAL heuristic overall, and is easier to implement as well. Our results show how difficult it is to correctly estimate how much garbage there is in the database across diverse database structures and application behaviors. The problem of very accurate garbage estimation heuristics that work across different database types requires further investigation. In the remainder of our results, we show the performance of the SAGA policy using the GLOBAL heuristic because it was the most effective garbage estimation heuristic across both databases.

5.2 Efficiency of the SAIO and SAGA Policies

The results in this section attempt to show that the SAIO and SAGA collection rate polices result in efficient application performance, as measured by total I/O operations performed and amount of garbage reclaimed. To do this, we compare the performance of the policies against fixed-rate policies, where the rate chosen exactly corresponds to the desired performance. All the results in this section are based on averaging three simulation runs for each data point presented. In figures 9 and 10 we show how the GC I/O percentage and garbage percentage vary as a function of fixed collection rate in the two test databases. If these figures were available to the database user, and the user desired a particular GC I/O percentage or garbage percentage, then that user could choose the correct collection rate for that particular database. For example, if a user desired a garbage percentage of 30% in the ABT database, then they would choose a collection rate of approximately 130 pointer overwrites per collection.

In this section we compare the performance of our semi-automatic rate policies SAIO and SAGA, against an oracle that estimates the exact fixed rate required to produce the desired behavior. We call this policy the *fixed-rate oracle* (FRO). Our goal is to show that the SAIO and SAGA policies have performance that is very similar to FRO but require no such oracle knowledge.

Table 1 provides such a performance comparison. The table is broken into two parts, showing results from the ABT and OO7 databases. For each database, we show results for the SAIO policy and the SAGA policy using the GLOBAL garbage estimation heuristic. For each of these policies, we present results for a range of parameter settings (e.g., SAIO 2% corresponds to the SAIO policy with a setting of 2% GC I/O operations). For each parameter setting, we also show the performance

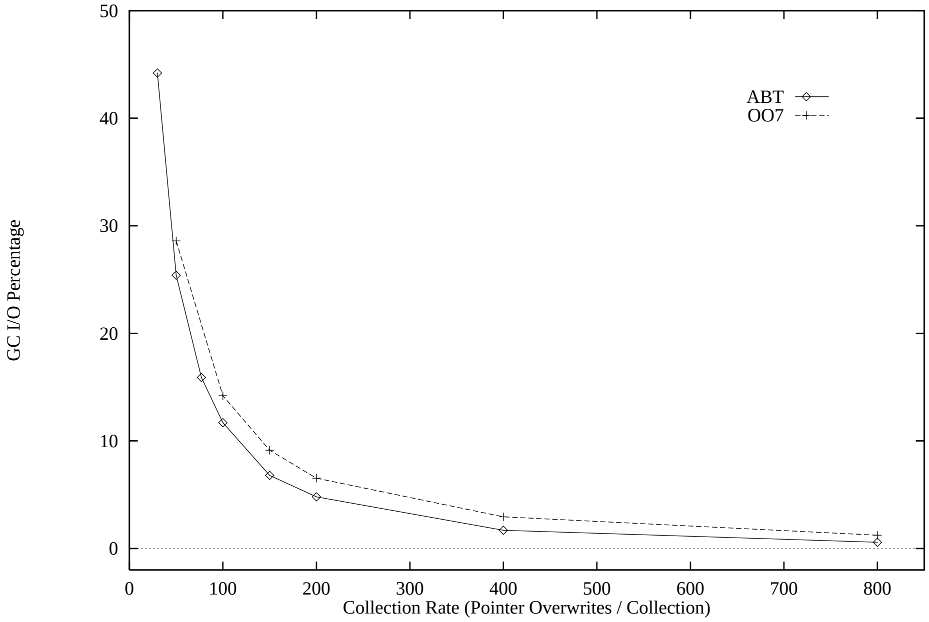

Figure 9: Percentage of Garbage Collection I/O Operations a Function of Collection Rate.

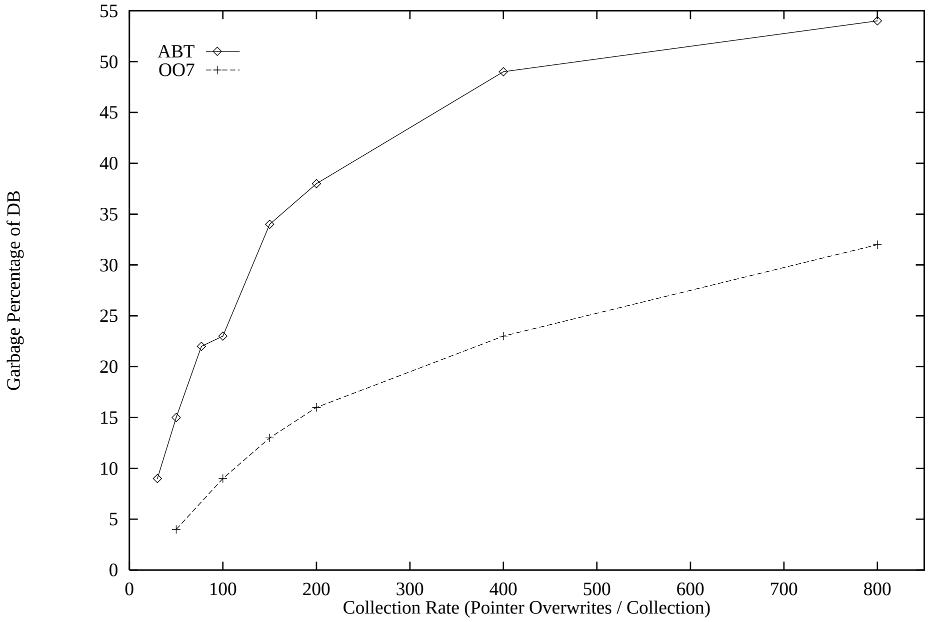

Figure 10: Percentage of Database that is Garbage as a Function of Collection Rate.

Collection Rate Policy	Rate (Pointer Overwrites/Coll.)	Total I/O Ops (App+GC)	GC I/O Ops as % of App I/O Ops		Amount Collected (KBytes)	Amount of Garbage as % of DB	
	Mean	Mean	Mean	Std Dev	Mean	Mean	Std Dev
ABT							
FRO(IO 2%)	290	35500	2	–	3800	42	–
FRO(IO 5%)	170	34760	5	–	4800	37	–
FRO(IO 10%)	110	34770	10	–	5800	24	–
FRO(IO 15%)	75	36000	15	–	6700	22	–
SAIO 2%	407	36263	1.9	0.15	2643	48	1.0
SAIO 5%	187	34920	5.0	0.31	5000	39	3.0
SAIO 10%	111	35268	9.9	0.14	5664	31	4.4
SAIO 15%	80	35700	14.5	0.33	6374	23	3.0
FRO(GA 10%)	35	42000	40	–	6800	10	–
FRO(GA 20%)	70	36500	17	–	6100	20	–
FRO(GA 30%)	130	34550	7	–	5300	30	–
SAGA 10%	51	37040	22.5	2.13	6627	15	1.0
SAGA 20%	138	34625	8.2	1.91	5364	31	2.3
SAGA 30%	433	35951	1.9	0.38	2832	48	0.6
OO7							
FRO(IO 2%)	600	31100	2	–	1600	28	–
FRO(IO 5%)	275	31900	5	–	2300	18	–
FRO(IO 10%)	130	33000	10	–	2700	11	–
FRO(IO 15%)	90	34900	15	–	2900	8	–
SAIO 2%	685	31723	1.6	0.07	1015	31	0.0
SAIO 5%	414	31406	2.7	0.07	1471	28	0.0
SAIO 10%	212	31934	5.5	0.05	2054	22	0.0
SAIO 15%	134	33030	8.3	0.09	2275	20	0.0
FRO(GA 10%)	110	33800	12	–	2800	10	–
FRO(GA 20%)	300	31800	4	–	2200	20	–
FRO(GA 30%)	690	31300	2.5	–	1400	30	–
SAGA 10%	150	33202	9.2	0.13	2799	9	0.6
SAGA 20%	251	31307	5.2	0.13	2438	19	0.0
SAGA 30%	330	31342	3.7	0.07	2333	24	0.0

Table 1: Comparison of Semi-Automatic Collection Rate Policies and the Corresponding (estimated) Fixed-Rate Values.

of the FRO policy, where the rate was obtained by reading the rate corresponding to the parameter from the previous figures. Thus, the FRO(IO 2%) row corresponds to and should be compared with the SAIO 2% row.

The table shows several things. First, if we compare the fixed-rate policies in the ABT and OO7 databases, we see that the correct fixed rate (from the second column) is highly application dependent. For example, to achieve a 2% GC I/O rate in ABT, we must collect every 290 pointer overwrites whereas in OO7, we must collect every 600 pointer overwrites. Likewise, to achieve 10% garbage in the ABT database, we need to collect every 35 pointer overwrites, whereas in the OO7 database, we would only collect every 110 pointer overwrites.

Next, we see that the results from the figures in Section 5.1 are reproduced when multiple runs are performed. Thus, in the ABT database, the SAIO heuristic is very accurate at estimating the GC I/O percentage requested (shown in the fourth column), and the SAIO average collection rates (show in the second column) correspond very closely to the FRO collection rates a user would have chosen. The total I/O performance of the SAIO policy is also very close to that of the FRO policy. In the OO7 case, the SAIO policy is not as accurate, as mentioned above, but the I/O performance of the policy is always comparable or better than that of the FRO policy.

For the SAGA policy, we see that the policy overshoots the requested value in the ABT database (from the second to last column) but is quite accurate or slightly undershoots the requested value in the OO7 database. Again, for the SAGA policy, the total I/O rates of the policy are typically comparable or better than the corresponding FRO policy. Our conclusion from this table is that chosing the best fixed-rate policy is database dependent, and our semi-automatic collection rate policies are effective at determining a proper collection rate and result in overall performance that is comparable to the best fixed-rate policy possible.

Finally, the small standard deviations in the table indicate that our results are consistent and reproducible.

5.3 Scalability

In this section, we investigate how effective our collection rate policies are in databases of different sizes. A major advantage of our semi-automatic policies becomes clear when the database size grows. Even if figures 9 and 10 were available to a database user for a database of one size, what fixed rate should be chosen if the size of the database grows by a factor of ten? In this section, we present results from the SAIO and SAGA policies in ABT[2] databases of different sizes and also show one attempt at scaling the FRO values we discussed in the previous section. Except for the smallest database, whose values are means of three runs, results in this section are computed from one simulation run for each point shown.

Figures 11 and 12 show how the SAIO and SAGA policies scale to databases of larger sizes. For these results, we scaled the partition size as the database grew, resulting in roughly the same number of partitions in databases of all sizes. Figure 11 shows the SAIO policy with target GC I/O percentage set at 10% across a range of database sizes up to 22 megabytes. We also show the performance of an attempt to scale the "best" 10% I/O fixed-rate policy determined in the previous section. In particular, we determined the collection rate to be 110 pointer overwrites per collection. Our method for scaling the fixed rate with database size was to scale the fixed-rate by the same factor that we scale the partition size. If the fixed rate is not scaled in this way, then the results are uniformly worse. From the figure, we see that the SAIO policy always remains closest to the requested percentage of garbage for all database sizes, while the scaled fixed rate wanders both below and above the requested rate.

Figure 12 shows the SAGA policy with target GC I/O percentage set at 20% across a range of database sizes up to 22 megabytes. We also show the performance of an attempt to scale the "best" 20% garbage fixed-rate policy determined in the previous section. The fixed-rate policy is again scaled by the partition size for these results. In this case, we see that while the SAGA heuristic deviates from the requested value at the smallest database size, its accuracy actually improves as the database size is increased. At the same time, the scaled fixed-rate policy, which is the the FRO estimate for the smallest database, worsens in performance as the database size increases. From these figures, we conclude that our SAIO and SAGA policies are robust to changes in database size, while attempts to scale fixed-rate policies do not work well.

[2] Scalability results for OO7 are currently being collected and will appear in the final version of the paper.

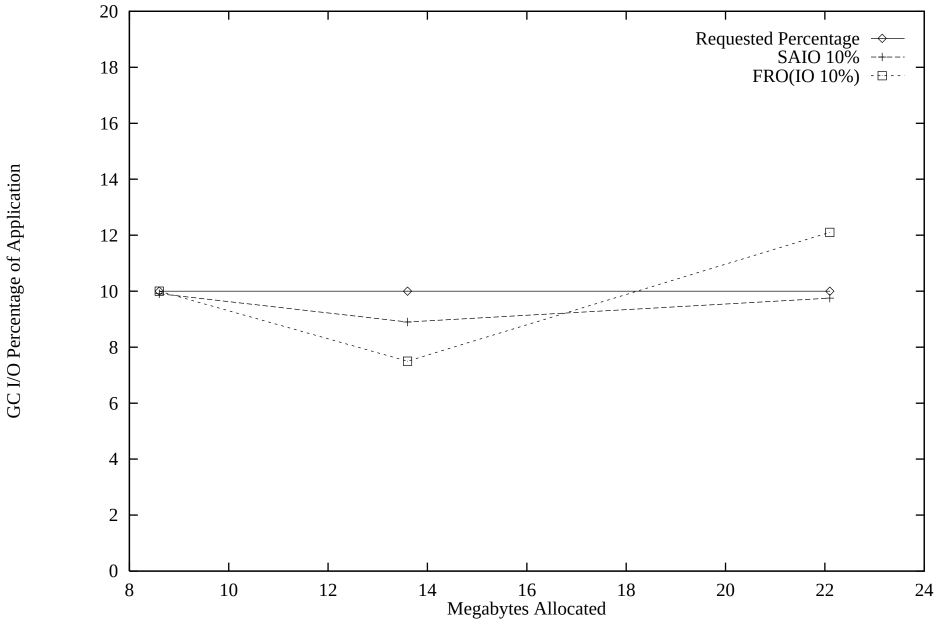

Figure 11: SAIO Collection Rate Accuracy as a Function of Database Size.

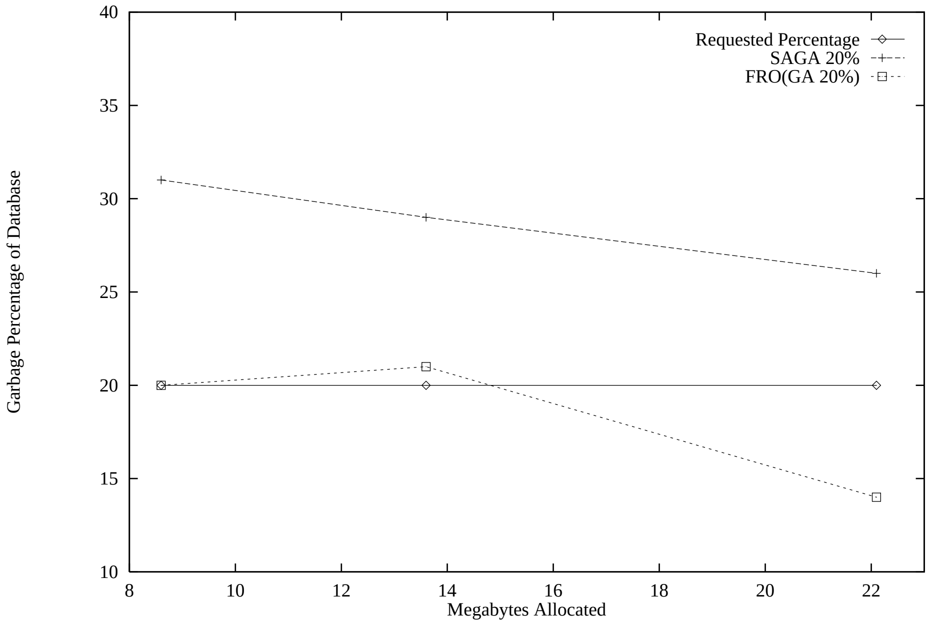

Figure 12: SAGA Collection Rate Accuracy as a Function of Database Size.

6 Summary

One very 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. Furthermore, the proper collection rate is a function of user preferences, database structure, application behavior, and database size. As a result, ad hoc techniques to determine a fixed collection rate fail. Furthermore, because application behavior can vary over time, no particular fixed collection rate may result in the desired performance. Finally, no previous work has concentrated on the difficult problem of setting 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 input from the database user about what level of performance the user desires. 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 application 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 changes in the application behavior over time.

We evaluated our proposed policies in the context of two very different object databases: ABT, an augmented binary tree database [9], and OO7, a more highly-connected benchmark object database [4]. Both of these databases have been used in previous object database garbage collection studies [9, 20]. Our results show that our SAIO policy is very effective at achieving the user-specified GC I/O percentage. Furthermore, the SAIO policy shows overall performance comparable to a fixed-rate policy, where the rate is chosen by an oracle.

The SAGA policy is more difficult to implement accurately because it depends on correctly estimating the current amount of garbage in the database. We investigated three heuristics for garbage estimation and concluded that a heuristic based on a global estimation of garbage per overwritten edge performs most accurately. We showed that the resulting SAGA policy is also quite accurate in controlling the amount of garbage in the database with excellent overall performance.

Finally, we showed that both the SAIO and SAGA policies scale to larger databases, whereas fixed collection rates obtained in smaller databases cannot easily be scaled to larger databases.

Acknowledgments. Artur Klauser was partially supported by BMfWF Kurt-Gödel scholarship 558.012/22-IV/A/a/94.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

[7] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[8] 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.

[9] 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*, page unknown, Minneapolis, MN, March 1994.

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

[11] Franz Incorporated. *Allegro Common Lisp User Guide*, Release 3.0 (beta) edition, April 1988.

[12] E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[13] 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.

[14] 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.

[15] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Comm. of the ACM*, 26(6):419–429, June 1983.

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

[17] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, Stanford, CA, February 1988. Also appears as tech report CSL-TR-88-351.

[18] 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.

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

[20] 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.

[21] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, Berkeley, CA, November 1989. Also appears as tech report UCB/CSD 89/544.