Evaluating Garbage Collectors for Large Persistent Stores*

Laurent Amsaleg
INRIA Rocquencourt
Laurent.Amsaleg@inria.fr

Paulo Ferreira
INRIA Rocquencourt
Paulo.Ferreira@inria.fr

Michael Franklin
University of Maryland
franklin@cs.umd.edu

Marc Shapiro
INRIA Rocquencourt
Marc.Shapiro@inria.fr

Abstract

We present some results and raise some issues regarding benchmarks for evaluating garbage collection in persistent distributed stores. We believe that such issues are relevant due to increasing research activity in this domain. Therefore, defining widely accepted benchmarks to evaluate the performance and impact of garbage collection solutions is urgent. Currently, the performance of garbage collectors is measured using synthetic, often unportable and/or controversial benchmarks, making the relative evaluation of each solution almost impossible. In addition, there is a lack of knowledge on the behavior of real applications, e.g., amount of garbage, rate of death, etc. This paper outlines our experience on benchmarking our prototypes. We also report what we think is relevant to measure and which missing information we would like to obtain.

1 Introduction

It has long been recognized that explicit storage management (e.g., *malloc()* and *free()*) increases code complexity and is highly error-prone, raising the risk of memory leaks and dangling pointers. For these reasons garbage collection (GC) for programming languages has long been an active area of research [Wil92].

Persistent Distributed Stores provides further motivation for the need for GC. The sharing of data and long-term persistence that are offered by these systems increases the potential damage caused by storage management errors, as mistakes made by one program can affect others. In addition, GC is needed to support *persistence by reachability* [ABC+83, ZM90]. By traversing the objects graph starting from some *persistent roots*, the collector is able to distinguish reachable (live) objects from unreachable (garbage) objects which can then be safely reclaimed.

In this paper, we briefly describe two GC solutions, one for Object Oriented Database Management Systems (OODBMS) and another for Persistent Distributed Shared Memory Stores. Details on these algorithms can be found in [AFG95] and [FS94, FS95]. Our main goals in designing GC algorithms was to make them unobtrusive, scalable, safe, and, as far as possible, complete. Within a persistent environment, such a goal renders traditional GC approaches incorrect (they eventually reclaim live objects) and/or inefficient.

*OOPSLA'95 Workshop on Object Database Behavior, Benchmarks, and Performance, Austin TX (USA), October 1995. http://info.acm.org/sig_forums/sigplan/oopsla/oopsla95.html

The increasing research activity in garbage collectors for persistent stores makes the task of defining a standard benchmark to evaluate each solution more urgent. Current benchmarks are artificial and/or controversial, making such evaluations difficult. In addition, there is a lack of knowledge on the behavior of real applications, in terms of creating garbage objects. In this paper, we report our experiences gained from measuring our prototypes, and use this experience to identify those missing measurements that we think are relevant.

2 Garbage Collecting Client-Server OODBMS

We have developed a server-based GC designed to work in the specific context of client-server OODBMS [AFG95]. We consider page servers that provide efficient and flexible features like inter-transaction caching, write-ahead-logging (WAL) based recovery, "steal-based" management of clients' buffers and 2-phase page-level locking. This flexibility, while providing opportunities for improved performance, also raises potential problems for garbage collection algorithms. To validate our ideas, we integrated (and measured) our scheme in the EXODUS storage manager [Gro93].

Our garbage collector has the main following characteristics:

- It is based on a Mark and Sweep approach. This approach can be more efficient and easier to integrate into an existing OODBMS. For example, the clustering produced by a copying-based GC may be in conflict with database requirements or user-specified hints.
- It is server-based, but requires no callbacks to clients. Our solution enforces its safety by beeing conservative with respect to transactions' effects.
- It is incremental and non-disruptive. It holds no locks on data and performs only minimal synchronization with client transactions. It has been designed to be efficient with respect to I/O.
- We use a partitioned approach in order to avoid having to scan the entire database before reclaiming any space.
- It works in the context of ACID transactions with standard implementation techniques such as two-phase locking and write-ahead-logging.
- It is fault-tolerant but requires minimal logging — crashes of clients or servers during garbage collection will not corrupt the database.
- It can co-exist with performance enhancements such as inter-transaction caching at clients and "steal" buffer management between clients and servers.

3 Garbage Collecting Persistent Distributed Shared Memory Systems

In the Larchant project, we designed and implemented a distributed shared memory (DSM) with tracing garbage collection in order to provide persistence by reachability (PBR) in a large-scale distributed system [FS94, FS95]. Within a general model of DSM, we specified a distributed tracing GC algorithm that scales, collects cycles, and is orthogonal to coherence.

Our garbage collector has the main following characteristics:

- It supports not only a mark-and-sweep but also a copying algorithm that enhances the clustering between objects.
- We have adopted a partitioned approach where subsets of the memory are collected independently.
- Collectors of different replicas of the same partition running at different sites do not synchronize with each other and do not imply the execution of any DSM coherence protocol event. Moreover, our GC works even if every replica of the same object are not all up-to-date.
- Full asynchrony between GCs is achieved by being conservative with respect to applications updates and requires messages to be delivered in causal order.
- For collecting cycles of garbage that span several partitions, groups of partitions mapped at some site are collected simultaneously. The choice of the group to collect is heuristic, and aims at maximizing the amount of garbage reclaimed while minimizing the cost.

4 Benchmarking Garbage Collectors for Persistent Systems

We report here several factors we think are important to measure. We isolated two main categories of parameters: the first one helps to understand the basic characteristics of the collector, like the elementary costs of the basic components of the algorithm. The second one gives an idea of the behavior of the global system when user programs execute concurrently with GCs.

4.1 Measuring the Cost of the Implementation

4.1.1 What to Measure?

- *Bookkeeping Overhead.* GC may need to manage specific data enforcing their correctness or fault tolerance aspects. This may be specially significant when some extra data need to be stored on stable storage, like log records.
- *Partitioning.* Piecewise collections of the object space improves performance. However, keeping trace of inter-partition references raises fault tolerant issues and requires extra bookkeeping overhead.
- *Garbage Detection and Reclaimation Speed.* The number of objects analyzed and the rate of space reclaimation, say, per second, may highlight some bottlenecks in the implementation of low-level features. These features may be, for example, the management of free space, colormaps or forwarders.
- *Synchronization.* Correctness of concurrent solutions often requires some costly synchronization like barriers, memory flips, callbacks or exclusive locks on data. Moreover, global synchronizations between multiple collectors may also be required in a distributed context.
- *Cleaning Side effects.* The cleaning of the database (DB) may in turn trigger some coherency protocols that try to maintain the consistency of some data stored at remote sites. These data may be multiple replicas of subsets of the DB or some GC specific data that deal with the distributed aspects of the environment. Some overhead may come from constraints on exchanging messages, like enforcing causal order or dealing with network failures.

4.1.2 What Influences these Measurements?

- *Clustering.* This factor has a direct impact on the number of I/O the collector does during a collection cycle. Varying the clustering of data gives the sensitivity of the *Garbage Detection* (marking or copying) phase of the collection. The influence of clustering is also tied to the *caching* of data at the site of collection.

- *Reclustering.* Some GC (like copying collectors) recluster objects, which in turn enhance the performance of the system (including the performance of the collector itself). Moreover, copying GC reduces DB fragmentation while Mark and Sweep GC do not (except when slotted pages are used by the DBMS). However, moving every live object can complicate recovery.

- *Amount of Garbage and Size of the Database.* We need to evaluate how the *speed* of GC scales varying the amount of garbage and the size of the database. Increasing the garbage % decreases the number of live objects that need to be traversed. On the other hand, it increases the volume of space that need to be freed. The size of the database — or of each partition of the DB — directly influences the duration of each collection cycle.

4.1.3 How to Get these Measurements?

Synthetic benchmarks are helpful in understanding the basic costs of a given implementation. These benchmarks should be very simple to be easily mastered. They should also enable us to play easily with the above factors to clearly see the effects of their variations.

Our Experience

To measure our client-server OODBMS garbage collector, we designed a synthetic benchmark consisting of simple linked-lists of objects. Database pages were fully packed with objects, however, we varied the percentage of garbage objects *in each page* as an experimental parameter. We also varied the "clustering factor" of objects in pages. This factor determines the number of live objects on a page that the marker can scan before crossing a page boundary. For example, with "1/2 Clustering", half of the live objects in a page can be traversed before a page boundary is crossed.

Such a benchmark enabled us to easily get the effect of garbage percentage, clustering and partition size variations. Our collector scales linearly with the partition size for all % garbage values. Collecting 80 Mb where 50% of the objects in each page are garbage reduces the response time of the marking phase by 10% with respect to the no garbage case. The sweeper performance is virtually independent of the percentage of garbage in a page. As shown by our curves, the major overhead for sweeping is fetching the page from disk, the freeing of garbage objects beeing insignificant (pure CPU cost). The influence of the clustering of objects convinced us that we need to study more complex graph traversals than the traditional depth-first or breadth-first schemes.

Our simple benchmark was also useful to isolate the cost of bookkeeping for correctness (from 1 to 8 %) and fault-tolerance (a 80 bytes long extra log record for every swept page) or the effect of several optimizations that were aimed at reducing I/Os.

However, this synthetic benchmark was not fully satisfactory. For example, because we vary the percentage of garbage objects *in each page*, the collector reads the same number of pages regardless of the amount of garbage in the pages; thus, only the performance costs, but not the benefits of garbage collection (i.e., reduced I/O) were measured.

4.2 Performance of Global System

4.2.1 What to Measure?

- *Response Time of User Transactions and Throughput of the System.* It is fundamental to know the impact of GC on the performance of user programs. Two factors may slowdown user applications: the extra load experienced by the system and the potential synchronizations with the collector(s).

- *Slowdown of the Collector.* Besides measuring the slowdown of user computations, we must know the impact of programs on a concurrent GC.

4.2.2 What influences these Measurements?

- *Clustering and Reclustering.* Clustering has a significant impact of the performance of both the collector and the applications. However, the (re)clustering produced by a copying-based collector may be in conflict with database requirements or user-specified hints (e.g., [BD90, GA94]). For example, generational collectors tend to cluster objects based on their age. After a GC, objects must still be clustered with respect to application needs. No implemented copying-based GC follows this fundamental requirement.

- *Caching and Prefetching.* Concurrent execution of the GC and user programs impact caching. If their respective access patterns differs completely, the system may trash due to a low cache-hit ratio and intensive I/Os. In contrast, one may prefetch the pages the other needs if they have similar access patterns. Obviously, caching and clustering are strongly related, each being able to reduce the negative effects of the other.

- *Effect of Scheduling.* The negative impact on user programs performance can be traded off against the execution time of the collector by varying the aggressiveness with which the collector is scheduled. Favoring user programs will reduce their slowdown when the collector is running. However, this slowdown will be incurred over a longer time period, as the collector will take longer to complete its job. Furthermore, slowing down the collector can hurt performance by impacting its ability to quickly free up wasted space.

- *Workload.* The load of the system directly impacts GC performance. Inactive (or less intensive) periods may be preferably used by the collector.

- *Choosing the Optimal Partition.* Randomly choosing the partition to collect may not be a good idea. Some heuristics that evaluate the volume of garbage generated by reference updates may improve performance (e.g., [CWZ94]).

- *Choosing the Right Collection Frequency.* Rates of GC directly impact performance. Too frequent or too rare collections both have a negative effect on performance. Smart policies driven by the behavior of applications may help (e.g., [CKWZ94]).

4.2.3 How to Get these Measurements?

Because we are measuring the impact of GC on applications, "real-world" benchmarks seem to be useful. Consequently, some collectors are evaluated through the use of an extended version of a standard database benchmark such as 007 [CDN93]. The extension usually consists in some temporal evolution of the contents of the database, i.e., creating new objects and/or generating

some garbage. However, what drives this evolution is often random values choosen in between arbitrary boundaries. Aside these potentially controversial settings, understanding the numbers given by the performance study is even more complicated than in the synthetic benchmark case. With such benchmarks, the relevant factors are swamped by the mass of irrelevant details.

Our Experience

We also used our simple synthetic benchmark to see the effects of GC on the performance of applications. Its artificial nature can not reflect the slowdown any real application would pay. Our benchmark revealed to be inadequate to measure the influence of some major factors. For example, we were unable to vary the load of the system or the clustering of objects in a realistic way.

4.3 Can we Define a "Standard Benchmark"?

We raise here several questions that should be solved before defining a standard benchmark for evaluating garbage collection schemes for persistent systems. We think we need (at least) the following information before defining such a benchmark.

- At what rate does garbage appear?
- What is the distribution of garbage objects in the database?
- Is garbage creation a steady or sudden phenomenon?
- Among all reference updates, how many generate some garbage?
- Can we evaluate the average sharing of each object? Simpler schemes may be used for non-shared data!
- Do applications clustering needs conflict with the collector reclustering process?
- Garbage collection is costly. However, this cost benefits in turn sooner or later to applications. GC reduces the number of I/Os applications will perform in the future because it reduces the size of the database, data fragmentation, and may recluster objects. Measuring these differed enhancements is very difficult.

5 Conclusion

Benchmarking GC for persistent systems is difficult. First, the implementation of the collector must be evaluated to know the costs of its low-level details. Synthetic benchmarks are relevant to get this first set of numbers. Second, the impact of GC on user applications must also be know to evaluate the performance of the global system. In this case, more realistic benchmarks are required.

However, the lack of knowledge on the behavior of applications in terms that interest GC is a difficult obstacle. Before defining such a "real-world" benchmark, it may be helpful to monitor several major database applications. This monitoring will certainly give us some empirical knowledge on their garbage generation behavior and temporal evolution. This knowledge may lead us to classify database applications in several groups, each having different GC needs and requirements. Instead of defining a single and universal benchmark for persistent GC, we may find relevant to have multiple benchmarks, each tailored to reflect the interactions between the GC and the main characteristics of a given class of application.

References

[ABC+83] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, 1983.

[AFG95] L. Amsaleg, M. Franklin, and O. Gruber. Efficient Incremental Garbage Collection for Client-Server Object Database Systems. In *Proc. of the 21th VLDB Int. Conf.*, Zürich, Switzerland, September 1995.

[BD90] V. Benzaken and C. Delobel. Enhancing Performance in a Persistent Object Store: Clustering Strategies in O_2. In *4th Int. Workshop on Persistent Object Systems*, pages 403–412, Martha-Vineyard, Massachusetts, September 1990.

[CDN93] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. In *Proc. of the ACM SIGMOD Int. Conf*, pages 12–21, Washington D.C., May 1993.

[CKWZ94] J. Cook, A. Klauser, A. Wolf, and B. Zorn. Effectively Controlling Garbage Collection Rates in Object Databases. Technical Report CU-CS-758-94, University of Colorado at Boulder, Boulder, Colorado, October 1994.

[CWZ94] J. Cook, A. Wolf, and B. Zorn. Partition Selection Policies in Object Database Garbage Collection. In *Proc. of the ACM SIGMOD Int. Conf.*, pages 371–382, Mineapolis, Minnesota, May 1994.

[FS94] P. Ferreira and M. Shapiro. Garbage Collection and DSM Consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–241, Monterey, California, November 1994. ACM.

[FS95] P. Ferreira and M. Shapiro. Garbage Collection in the Larchant Persistent Distributed Shared Store. In *5th Workshop on Future Trends in Distributed Computing Systems (FTDCS '95)*, Cheju Island (Korea), August 1995.

[GA94] O. Gruber and L. Amsaleg. *Object Grouping in Eos*, pages 117–131. In [ODV94], May 1994.

[Gro93] EXODUS Project Group. EXODUS Storage Manager Architectural Overview, 1993.

[ODV94] T. Özsu, U. Dayal, and P. Valduriez. *Distributed Object Management.* Morgan-Kaufman, San Mateo, California, May 1994.

[Wil92] P. Wilson. Uniprocessor Garbage Collection Techniques. In *Int. Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–43, St. Malo, France, September 1992. Springer-Verlag.

[ZM90] S. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, California, 1990.