The Memory Behavior of the WWW, or The WWW Considered as a Persistent Store

Nicolas Richer and Marc Shapiro

INRIA - SOR group, BP. 105,
78153 Le Chesnay Cedex, France
{Nicolas.Richer, Marc.Shapiro}@inria.fr
http://www-sor.inria.fr/

Abstract. This paper presents the performance evaluation of five memory allocation strategies for the PerDiS Persistent Distributed Object store in the context of allocating two different web sites in the store. The evaluation was conducted using (i) a web gathering tool, to log the web objects graph, and (ii) a PerDiS memory simulator that implements the different allocation strategies. Our results show that for all the strategies and parameters we have evaluated, reference and cycle locality are quiet poor. The best policy seems to be first sequential fits. Results are linear with the size of the garbage collection Unit (a bunch). There is no clear optimum, but 64K to 128K appear to be good choices.

1 Introduction

PerDiS [FSB+98] is a new technology for sharing information over the Internet: Its abstraction is a Persistent Distributed Object Store. PerDiS provide a simple, efficient and safe abstraction to the application programmer. To attain this goal, it is necessary to understand the way target applications use the persistent store and how the different mechanisms and heuristics perform. This study is the target of task T.C 3 of the PerDiS project. This paper presents the results we have obtained.

The PerDiS persistent store is logically divided into clusters. A cluster is the naming and security unit visible by the programmer. A cluster is attached to a "home" site[1] but can be accessed from any site. From the garbage collection point of view, cluster is further subdivided into bunches, i.e. garbage collection Units. The Garbage Collector always runs on whole bunch and this unit will be completely replicated at one site in order for the GC to run on[2]. This means also that bunches will be stored in a contiguous area in memory. A reference inside a bunch is cheaper than between two bunches (an inter-bunch reference). The latter needs using stubs and scions. The fewer references will be inter-bunch, the better the PerDiS system will perform.

[1] The home site ensures data security and reliability.

[2] For latency reasons, bunches could be divided in pages but the page forming a bunch should be fully located at a site for the Garbage Collection to operate on.

1.1 What Are Garbage Collection Components

A Garbage Collection components is a set of objects reclaimed by the garbage collector all at the same time. This could be a single object or several objects link together in a way that they become unreachable all at the same time. Typically, this can be a set of objects with circular references between them.

In this context, Strongly Connected component is an approximation for Garbage Collection component. Rigorously, Garbage Collection components are Strongly Connected component plus all the objects reachable only from this components, but we don't make the difference in this study, because, unfortunately, we are not able to extract real Garbage Collection components from an object graph yet. This approximation may grow up the overall proportion of objects and bytes included in Garbage Collection components. Studying Strongly Connected component provide only minimum proportions, and we don't currently know the maximums.

Note also that Garbage Collection components are often called cycles in this paper because it's shorter.

1.2 The PerDiS Garbage Collector

The PerDiS Garbage Collector use a reference counting algorithm to handle inter-bunches references, this to avoid costly distributed Garbage Collection Algorithms. In consequences, some garbage could not be reclaimed in some situation. This is the case of Garbage Collection components composed by several objects spread between different bunches. This kind of garbage could be reclaimed on the of all the bunch is containing the objects were run located at the same site. This is, unfortunately, the drawback of using reference counting.

The assumption we have made during the design of the PerDiS Garbage Collector is that such kind of unreclaimable components are sufficiently rare to be forgotten and we don't put any kind of global tracing collector for them. Of course, this assumption should be verified in practice and this is precisely why we are interested by Garbage Collection components in this study. Depending of our experimental results, the need for a global tracing collector mechanism will be reconsidered. The important point here is the fewer cycles are inter-bunches, the less garbage will remain unreclaimable. This suggest that object placement strategies (i.e. allocation algorithms) should minimize inter-bunch Garbage Collection components and inter-bunch references. A secondary goal will be to minimize also the number of scions and store fragmentation.

In the current PerDiS implementation, the initial allocator clustering is permanent, because PerDiS support the uncooperative C++ language that does not provide natively all the informations required to perform object relocation. In the future, we plan to implement object relocation in PerDiS using type information extracted by `typedesc` [Sal99], a tool we have specifically implemented for that.

A complete description of the PerDiS Garbage Collection Algorithm is available in [FS94,FS96]. [BFS98] describe more deeply the implementation.

1.3 Analysis Methodology

Our methodology is to record the persistent object graph during application runs, using a log mechanism. The log subsequently supports simulation of different strategies and parameters. We focus primarily on evaluating memory allocation and clustering strategies. Many studies of memory allocation and clustering strategies already exist [WJNB95,WJ98,ZG92a,ZG92b,DDZ93], but none has been conducted in the context of a persistent distributed garbage collected memory[3]. This is probably related to the very few number of existing persistent store, but even for those that exists, most of them rely on manual memory deallocation. For example, one of the most advanced distributed persistent stores, ObjectStore [LLOW91] is based on manual persistence, in contrast with persistence by reachability [ABC$^+$83] using a distributed garbage collector as in PerDiS.

1.4 Why Study the Web ?

Wilson in [WJNB95] demonstrates the low representativity of synthetically generated allocation requests. Therefore, we chose to evaluate the memory allocation and clustering strategies using inputs from real applications. Consequently, we need some significant persistent applications for our evaluation. Unfortunately, applications that use persistent objects store are rare, and a few that exists are not easily accessible. Furthermore, PerDiS is designed to provide easy access to persistent data over a geographically widely-distributed network. We need applications running in such a large-scale context. The Web application has attracted our attention because it is easily accessible and widely distributed. Hence our decision was to study the behavior of PerDiS as a distributed storage medium for web sites.

Practically, we have not actually stored the full web sites in the PerDiS memory, although it is perfectly feasible to do so[4]. Instead, we have used the PerDiS memory simulator. This simulator has been designed specifically to reproduce the PerDiS memory behavior against several allocation strategies and parameters.

2 Gathering the Web Object Graph

We briefly present in this section how we gather a graph of objects from a web server in order to study it. The basic gathering algorithm is a depth-first top-down traversal implemented using a stack:

1. Push some root document URL on the fetch stack.
2. Pop an URL from the fetch stack, retrieve the corresponding document, and parse it to extract the URL it contains.
3. Push all the contained URLs that are in scope on to the fetch stack.
4. Return to step 2 until fetch stack becomes empty.

Starting from an existing mirroring tool seemed a reasonable approach. We chose the w3mir[5] all purpose HTTP copying tool because its source code is freely available and we have good experience with it.

The web objects graph is recorded in the generic log format version 2 of our generic log handling library[6]. Since w3mir is written in PERL[7], we created a PERL interface for the generic log library.

We first integrated the recording of the web objects and references to w3mir by minimizing the modifications in the mirror algorithm in less than three days. The result was not acceptable as is because many processes and fetch was duplicated, so in a second step we have modified the w3mir algorithm to remove this.

The following problems, which are inherent to the World Wide Web, should be noted:

– A document's "last modified date" is often not correctly returned by web servers. When this occurs, we take the current date as the "last modified date".

[3] Data in persistent and transient memory is typically not the same. A lot of data in transient memory (used by graphical library for example), for intermediate results will never be persistent.

[4] QMW, a partner of the PerDiS project, did that for another purpose (`http://www.dcs.qmw.ac.uk/research/distrib/perdis/docs/perdisweb.html`).

[5] `http://www.math.uio.no/~janl/w3mir/`

[6] The documentation of this library is available from `ftp://ftp.inria.fr/INRIA/Projects/SOR/misc/analysis/doc/log_facilities.ps.gz`

[7] `http://www.perl.com/`

– Recording the evolution of the Web Objects Graph is difficult in practice. Since there is no reliable protocol for change notification, we need to keep a local full copy of the web site. Furthermore, many web sites are modified rarely; therefore measurements would have to be repeated over very long periods (maybe 6 months or more) in order to see significant modifications. Instead, the results presented in this paper concern a single snapshot of each studied web site.
– The graph that we obtain, using the mirroring method, is different from the real Web graph. Several kinds of document (CGI scripts, Java applets, Netscape Java-script, etc) could not be handled and furthermore, since documents of this types are sometimes used to provide the menu to navigate through the entire web site, a large part of this kind of web sites could not be visible. As far as we know, all existing indexing engines has the same problem with this kind of documents. Nevertheless our partial graph should be sufficient for evaluation purposes and we have to take care of this problem when we choose the web sites to study.

3 Characteristics of the Web Sites Studied

This initial study involves the two web sites: `http://www.perdis.esprit.ec.org/` and `http://www-sor.inria.fr/`. This choice was motivated by three criteria:

1. Network connection quality. Since we need to fetch the full web site contents each time we gather the graph, a very fast connection to the servers is crucial.
2. Prior knowledge. Since there are several kinds of documents that our web gathering tool can't handle, having some previous knowledge of the web site content should avoid choosing, for instance, a site where most of the document cannot bet considered because they are all only accessible from a Java applet menu.
3. Total Size. To be able to perform our simulations sufficient memory resources should be available on the simulation machine. In our current setup, this limits the total web site to approximatively 1 Gigabyte.

All the relevant characteristics of the two studied sites are presented in the next two sections.

3.1 Site 1: http://www.perdis.esprit.ec.org/

The PerDiS web site was studied on Thursday September 23 1999. At this date, it contained 3302 objects, with 31408 valid references[8] for a total size of 109 Mbytes.

The smallest object size is 0 byte and the largest 10 Mbytes. The median is 2.4 Kbytes, for an arithmetic mean of 33 Kbytes, and a variation coefficient of 9.7. We present the size frequency distribution in Fig. 1.

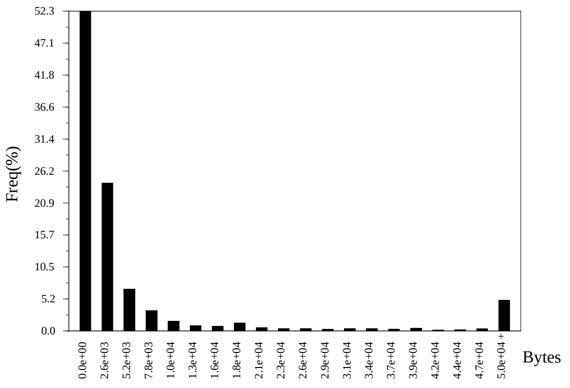

Fig. 1. PerDiS: object size class frequency distribution

Objects in the PerDiS web site are aged between 6 seconds old to a little more than 3 years. The median age is 1.5 years, for an arithmetic mean of 1.3 years and a variation coefficient of 0.59. We present the age frequency distribution in Fig. 2.

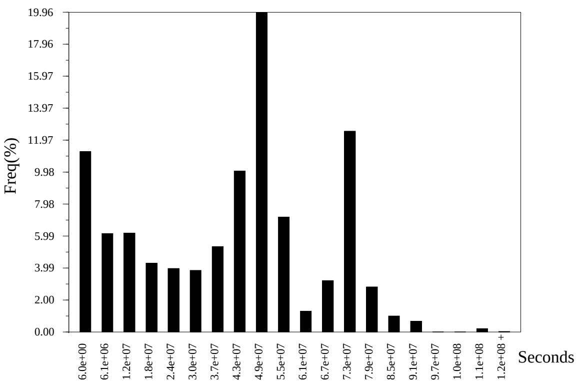

Fig. 2. PerDiS: object age class frequency distribution

[8] references that points to documents that exists, not including dangling references that point to unexisting documents (or protected documents that are not available for us).

The average density of references[9] is 9.5 per object and 0.28 per Kbyte.

There are 22 Strongly Connected components that contain 2284 objects (69.2% of total) for a total size of 15 Mbytes (13.5% of total). The smallest S.C component contains 3 objects (7 Kbytes) and the largest 1758 objects (13.6 Mbytes)[10]. Figure 3 shows the size frequency, in number of objects, of strongly connected components. This figure appear quite uniform because there is only 22 components for 19 size ranges, so there is only one component in each size range, except for the seventh bar where there is two and the first bar where there is three. Figure 4 shows strongly connected size frequency in bytes.

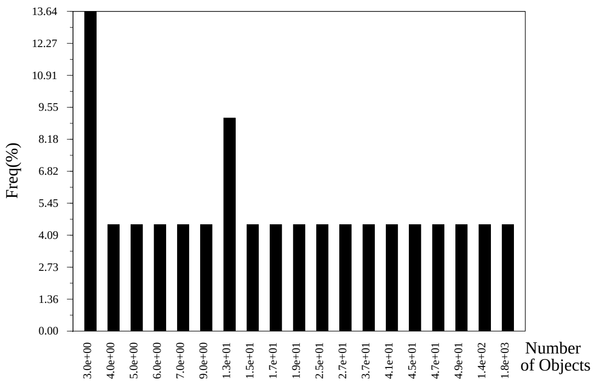

Fig. 3. PerDiS: size frequency, in number of objects, of strongly connected components

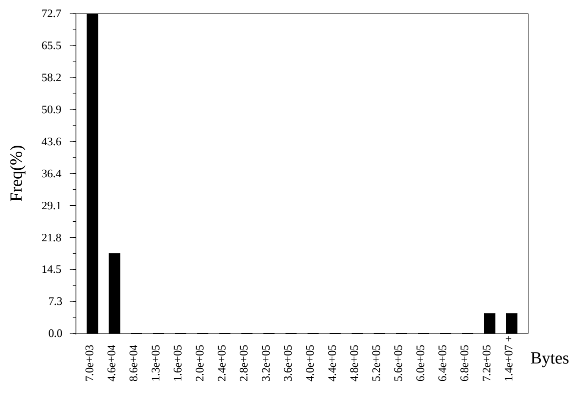

Fig. 4. PerDiS: size frequency, in bytes, of strongly connected components

3.2 Site 2: http://www-sor.inria.fr/

The SOR web site was studied on Thursday October 14 1999. At this date, it contained 8823 objects and 222714 valid references, for a total size of 277 Mbytes.

The smallest object is 8 bytes and the largest one is 8.4 Mbytes. Median is 5.3 Kbytes for an arithmetic mean of 32 Kbytes and a variation coefficient of 5.4. We present the size frequency distribution in Fig. 5.

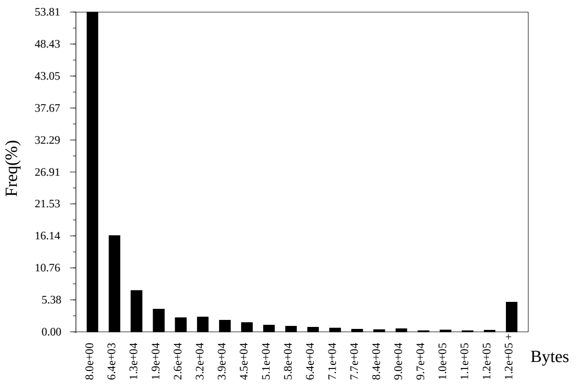

Fig. 5. SOR: object size class frequency distribution

Objects in the SOR web site are aged from 10 seconds to 4 years. Median age is 0.9 years, the arithmetic mean 0.8 years for a variation coefficient of 0.67. We present the age frequency distribution in Fig. 6.

The average density of references is 25.2 per object, and 0.78 per Kbyte.

There are 89 Strongly Connected components, containing 2819 objects (31.9% of total) for a total size of 40 Mbytes (14.56% of total). The smallest S.C component contains 2 objects (1 Kbytes) and the largest 381

[9] The density of reference is the number of references contained in one object, respectively in one Kilobyte.

[10] This component is in reality a mail archive with each message as an HTML document and this documents contained each a link to the previous and the next message.

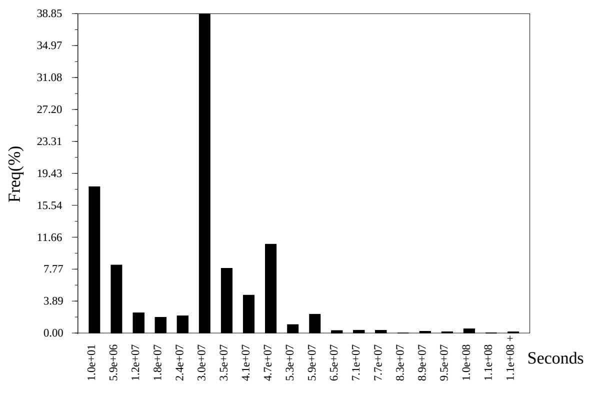

Fig. 6. SOR: object age class frequency distribution

objects (16.4 Mbytes). Figure 7 shows sizes frequency in number of objects, of the strongly connected components, whereas Fig. 8 shows their size frequency in bytes.

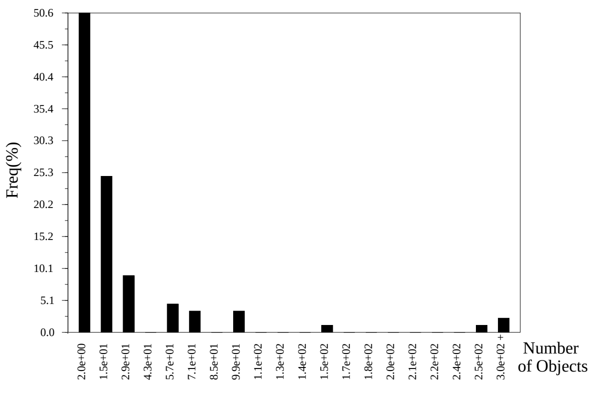

Fig. 7. SOR: size frequency, in number of objects, of strongly connected components

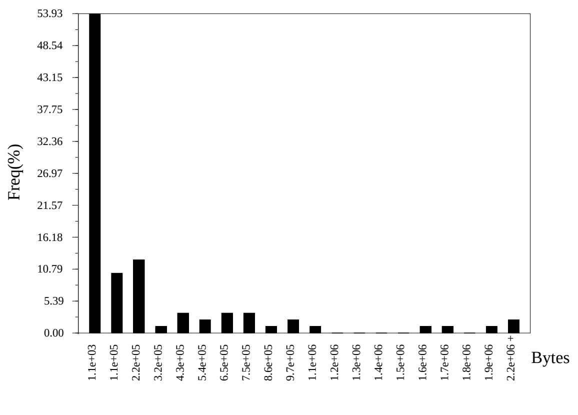

Fig. 8. SOR: size frequency, in bytes, of strongly connected components

4 Simulating Memory Allocation in the PerDiS Store

The PerDiS store simulator has been originally implemented to take a graph log, sort the memory allocation to simulate by document last modifications time, replay the allocations using the simulated allocation policy and finally generate a new log with all the old object locations in the heap replaced by the simulated new ones.

Sorting document by "last modifications time" introduced an error in the allocation replaying order because "last modifications time" returned by the web server is not reliable for every documents. Documents with unknown real modification dates exist and in this case, most of the web server return the current date as "last modifications time". To put a bound on the error introduced, we have measured the proportion of documents that have their "last modifications time" younger than the time the snapshot gathering begin. 4% of the documents are in this case in the PerDiS web site and 16% in the SOR web site. Other measurements on several other web sites shows a very high variability (2,3% to 100%) of this proportion.

The PerDiS store simulator is also able to replay the memory allocation in log order. According to the gathering algorithm we currently use (top-down depth-first traversal), log is ordered top-down depth-first.

5 PerDiS Memory Allocation Strategies

The current PerDiS API has applications explicitly place a specific object in a specific cluster. A cluster allocates an object in one of two kinds of bunches, depending on size. Under a certain threshold, it is allocated in a "standard" bunch of fixed size. Any layer object in a "special" bunch, containing only this object. This structure is based on the assumption that big objects are rare. This seems to be confirmed in our measurements (Fig. 1 for site 1 and Fig. 5 for site 2). In our simulations, the maximum proportion of large objects remain under 8% (with standard bunches of 32K).

Bunches are referenced in two doubly linked list, one for the standard bunches and one for the special ones. In the two following sections, we compare different strategies to allocate inside standard bunches and to select the standard bunch to allocate. Since standard bunch have a fixed size, we evaluate the allocation strategies using seven typical bunch size: 32K, 64K, 128K, 256K, 512K, 1024K and 2048Kbytes.

5.1 Strategies for Allocation Inside Bunches

In our PerDiS memory simulator, five strategies to allocate an object inside a bunch are currently implemented.

The *fixed size* strategy allows only allocation of object of the same size (or of the same size class) in the bunch. The free list uses a bitmap that keeps track of allocated and free blocks. This strategy is used in the PerDiS memory simulator in conjunction with the segregated bunches selection (see Sect. 5.2).

The other four use an address-ordered free list with deferred coalescing of the free block. The free list is doubly linked. When an allocation request can not be successfully completed because there is no large enough free block, contiguous free blocks in memory are coalesced and the free list is scanned again. There are four strategies to search for a free block in the free list:

- *first fit*: Walk through the free list from the beginning to first large enough free block, and allocate there.
- *best fit*: Walk through the whole list and allocate in the smallest large enough free block.
- *roving pointer*: Walk through the free list from the position where the last allocated object was taken from the free list.
- *linear allocator*: the same as *roving pointer* but never re-use the memory. This kind of allocator could be used only in conjunction with a copying garbage collection algorithm.

5.2 Strategies for Bunch Selection

The five strategies for selecting a standard bunch available in our simulator are all inspired by the strategies for allocating inside bunch:

- *First available bunch*: Bunches are stored in a list and allocation is done in the first satisfactory bunch.
- *Best available bunch*: Bunches are stored in a list and allocation is done in the best satisfactory bunch.
- *Last available bunch*: Bunches are also stored in a list but it is scanned from the last bunch where an allocation has been successfully completed.
- *Linear bunch*: Basically the same than *Last available bunch*, but never re-use the previous bunch in the list, allocate a new one. The same remark than for *linear* inside bunch allocation strategy apply.
- *Segregated bunch*: Use different bunches to allocate object of different size classes. Sizes are rounded up to the nearest power of two.

The first four strategies can be used in conjunction with *first fit*, *best fit*, *roving pointer* or *linear allocator* inside bunch allocation strategies. *Segregated bunch* can be used only with *fixed size* inside bunch allocation strategy.

6 Allocation Simulations Using Significant Strategies

For the simulations presented in this section, five allocation strategies have been selected. They correspond to the five available bunches selection strategies. The inside bunch allocation strategy is not significant in this set of simulations because only one single snapshot is considered and, by the way, there is no memory deallocation.

In the rest of the paper, the five strategies will be referenced using the following names:

1. *cluster_first*: allocate in the clusters using *First available bunch* strategy.
2. *cluster_best*: allocate in the clusters using *Best available bunch* strategy.
3. *cluster_roving*: allocate in the clusters using *Last available bunch* strategy.
4. *cluster_linear*: allocate in the clusters using *linear bunch* strategy.
5. *cluster_segregated*: allocate in the clusters using *segregated bunch* strategy.

6.1 Memory Allocation Simulations in Order of Last Modifications Date

In this section, the simulation results we present has been obtained using the five selected allocation strategies with data allocation ordered by last modifications date.

Site 1: http://www.perdis.esprit.ec.org/ Figure 9 shows the proportion of inter-bunch references for the five allocation strategies and for seven bunch sizes. Proportion of inter-bunch references is the percentage of all references that are to objects in a different bunch. Clearly *cluster_segregated* is the worse strategy from the

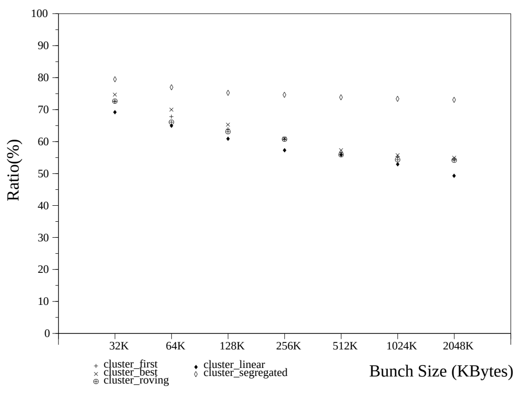

Fig. 9. PerDiS: proportion of inter-bunch references

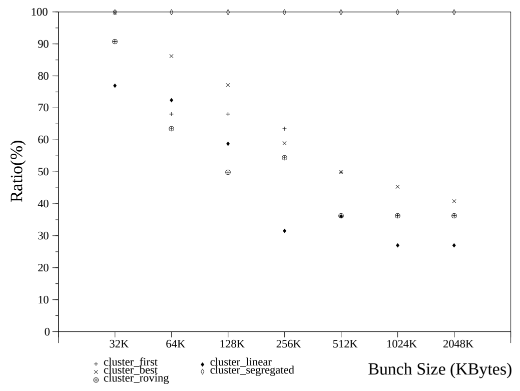

Fig. 10. PerDiS: proportion of inter-bunches cycles

point of view of reference locality but this is not a surprise. The best strategies from locality point of view (*cluster_linear* or *cluster_roving*, depending on bunch size) have proportion of inter-bunch references between 50%, for 2048K bunches, to 70%, for 32K ones, that is very high. The good locality results of the *cluster_roving* policy should be taken with care, because there is no freeing in our simulations and unfortunately, the roving policy is known to strew new objects among the old ones whose neighbors have been freed. The inter-bunches cycles proportion is the percentage of cycles which span bunches. Looking at this proportion, in Fig. 10, it appear that they are not more encouraging. *cluster_segregated* still give the worse results with 100% of inter-bunches cycles but the best strategy gives a proportion between 30% to 78% depends on bunch size. Figure 11 shows the number of scions, i.e. the number of objects that have incoming references from different bunches. This number is comparable to the number of objects for small bunches and go down by only 20% for large bunch and best allocation policy. This signify that, even in the best case, 80% of the objects have incoming references from different bunch, that is a high proportion.

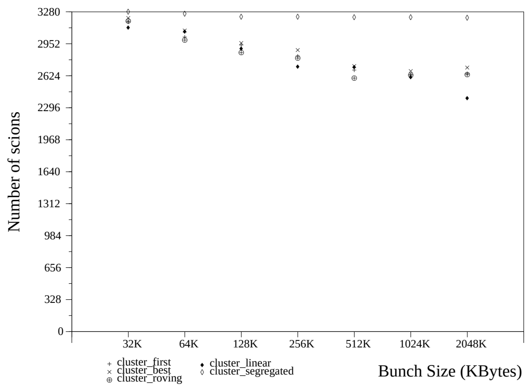

Fig. 11. PerDiS: number of scions

Finally, Fig. 12 gives an overview of memory fragmentation (internal and external combined) at the end of a snapshot simulation. This is the moment where the maximum of memory is allocated. Since there is no deallocation in our single snapshot, results about memory fragmentation are not significant, but they show that fragmentation increase with bunch size, as expected. They show also that policy known to be poor on fragmentation, the *cluster_segregated* strategy [WJ98], give the worse fragmentation result, even for small standard bunch sizes. This last result need to be confirmed by other simulations using several snapshots, representing the evolution (including deallocation) of the web site.

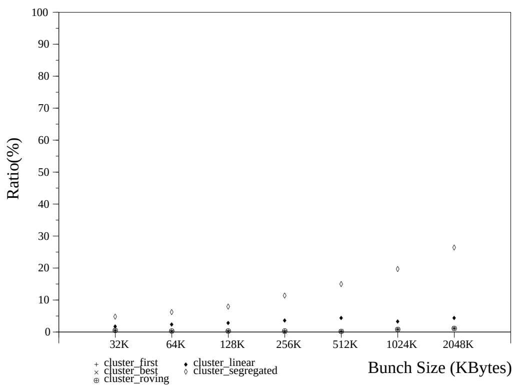

Fig. 12. PerDiS: memory fragmentation

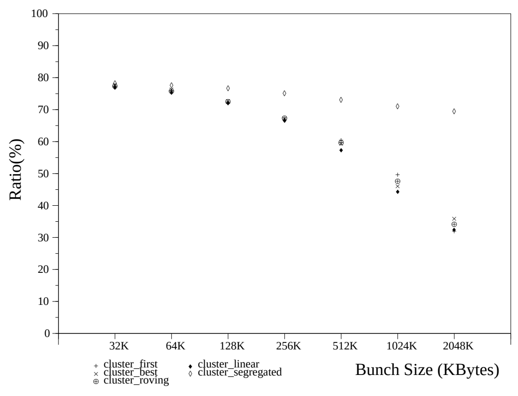

Fig. 13. SOR: proportion of inter-bunch references

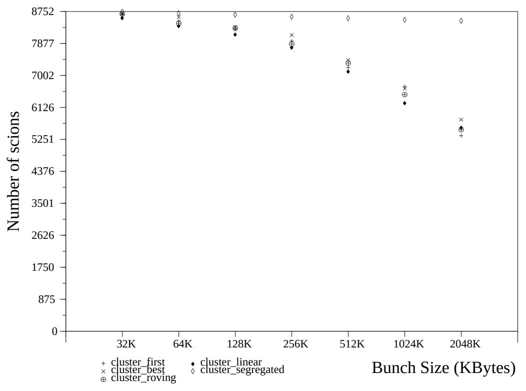

Fig. 15. SOR: number of scions

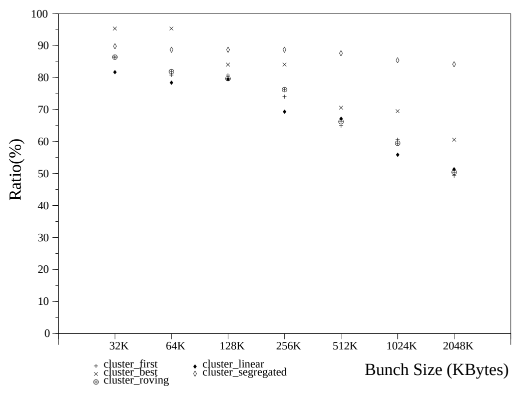

Fig. 14. SOR: proportion of inter-bunches cycles

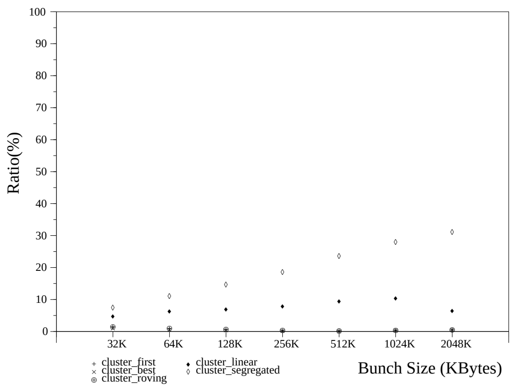

Fig. 16. SOR: memory fragmentation

Site 2: http://www-sor.inria.fr/ The second set of simulations, (on the second studied web site) shows many similarities with results for Site 1. Figure 13 shows the proportion of inter-bunches references. *Cluster_segregated* is still the worse strategy. The best proportion is between 30% to 79% for bunches respectively from 2048K to 32K. The inter-bunch cycles proportion, in Fig. 14, gives mostly the same results as for site 1, *cluster_segregated* is the worse and best proportion vary between 80% for 32K bunches to 50% for 2048K ones. Figure 15 show the number of scions. As for the PerDiS site, this number is comparable to the number of objects for small bunches, but go down more significantly (40%) for large bunch.

On the fragmentation front (Fig. 16), the tendency is almost the same as for site 1, except that the *cluster_linear* strategy gives worse results than *cluster_first*, *cluster_best* or *cluster_roving*. However, *cluster_segregated* is still the worse.

6.2 Memory Allocation Simulations in Depth-first Top-down Order

The simulation results presented here has been obtained using the same five selected allocation strategies as in Sect. 6.1 but with data allocation ordered by graph topology, in depth-first top-down order. For both PerDiS and SOR web site, we present only results about proportion of inter-bunch references and proportion of inter-bunch cycles as they are the most significant. Results about number of scions and fragmentation are omitted because they are almost similar to those presented in Sect. 6.1, using data allocation ordered by last modifications date and by the way there is not enough space in the paper for them.

Site 1: http://www.perdis.esprit.ec.org/ Figure 17 and Fig. 18 shows respectively proportion of inter-bunch references and proportion of inter-bunch cycles for the PerDiS web site. The proportion of inter-bunch references is 1% to 7% lower than the proportion we have using the last modifications date order, depend on the bunch size. For proportion of inter-bunch cycles, the difference is between 10% higher for 32K bunches to 10% lower for bunches of 1024K and 2048K.

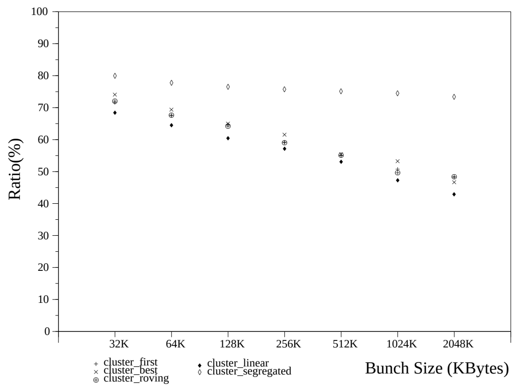

Fig. 17. PerDiS: proportion of inter-bunch references

Site 2: http://www-sor.inria.fr/ Figure 19 and Fig. 20 shows respectively proportion of inter-bunch references and proportion of inter-bunch cycles for the SOR web site. The proportion of inter-bunch references is 0% to 10% lower than the proportion we have using the last modifications date order, depend on the bunch size. For proportion of inter-bunch cycles, the difference is between 3% lower for 32K bunches to 15% lower for 2048K bunches.

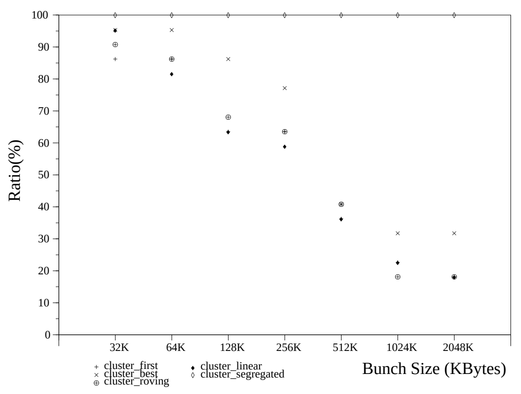

Fig. 18. PerDiS: proportion of inter-bunches cycles

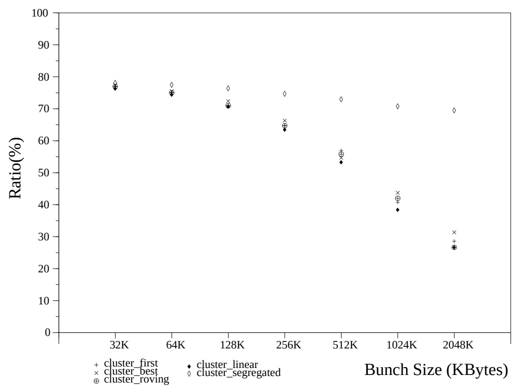

Fig. 19. SOR: proportion of inter-bunch references

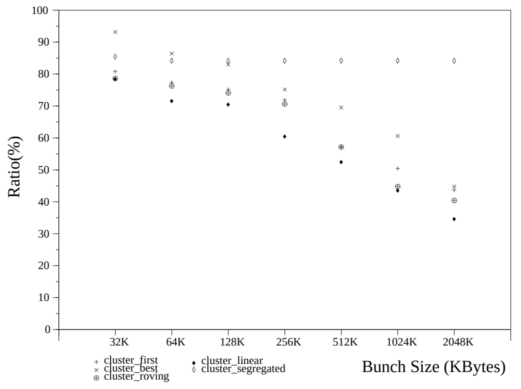

Fig. 20. SOR: proportion of inter-bunches cycles

7 Conclusion

In this paper, we have analyzed the intrinsic graph characteristics of two web sites (see Sect. 3) and evaluated the respective performance of five memory allocation strategies for a persistent store by simulating their behavior against the two web sites.

From the analysis of graph characteristics, we learned that most of the objects are small (around 80% are less than 50 Kbytes) and reference density is low. This confirms some known results. On the other hand, we learned that a large proportion of the objects is included in cycles (70% for site 1 and 30% for site 2) and the proportion of bytes involved is not negligible (12% to 14%). If we look at the characteristics of the individual cycles, it appears that most cycles are relatively small, both in number of objects and in bytes. This result contradict our assumption of a relatively small proportion of cycles. However, since the cycles are quiet small, this is not really a problem.

From the memory allocation simulations, we learned that a large proportion of references are inter-bunch: a large proportion of the cycles are inter-bunches whatever the bunch size and whatever allocation strategies we choose. This is not the results we expected. However these results need to be interpreted with caution because, first, we have studied only two web sites, which are not guaranteed to be representative of other web sites. Second, web application might not be representative of applications that use a persistent object store.

Finally, the simulation we have done with the allocation sorted by graph topology in depth-first top-down order shows that this order give always better locality results than the last modifications date order, but the difference is not so big. Most often, it stay between 5% to 10% and by the way using the allocation-order clustering policy seem to be feasible according to this set of simulations.

8 Future Work

This paper presented a first evaluation of the memory allocation strategies for the PerDiS Persistent Distributed Object Store. Future work direction are numerous.

One is to study the evolution of web sites over a long time period, by recording several snapshot until we are able to see significant graph modifications and deallocations. This is especially relevant for the measurements of memory fragmentation. Another direction is to implement more allocation strategies in our simulator, such as Buddy Systems [Kno65,PN77], Indexed Fits [Ste83], or Bit-mapped Fits [BDS91]. Extending our measurements on the web to many other web sites will be also interesting in the future. Another point is to study applications specifically targeting a persistent objects store and written in an object oriented fashion. More specifically, we plan to study the port to PerDiS of the Atlantis application from IEZ[11]. Comparing many other known dynamic re-clustering strategies (breath-first, depth-first, hierarchical decomposition, type directed, other ?) with our current allocation time placement strategy will be very interesting. In the more distant future, we may design and evaluate a new strategy where the specific target will be to minimize cross-bunches references. Those re-clustering strategies may be used in the PerDiS store if it appears that the static heuristics we have evaluated first are not efficient enough.

References

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proc. of the SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pages 157–164, Toronto (Canada), June 1991. ACM.

[BFS98] Xavier Blondel, Paulo Ferreira, and Marc Shapiro. Implementing garbage collection in the perdis system. In *Proceedings of the Eighth International Workshop on Persistent Object Systems*, August 1998. `http://www-sor.inria.fr/publi/IGCPS_pos8.html`.

[DDZ93] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, Dept. of Comp. Sc., Colorado University, Boulder, Colorado (USA), August 1993. `ftp://ftp.cs.colorado.edu/pub/cs/techreports/zorn/CU-CS-665-93.ps.Z`.

[FS94] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–241, Monterey CA (USA), November 1994. ACM. `http://www-sor.inria.fr/publi/GC-DSM-CONSIS_OSDI94.html`.

[FS96] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Proc. 16th Int. Conf. on Dist. Comp. Syst. (ICDCS)*, Hong Kong, May 1996. `http://www-sor.inria.fr/publi/LPRDSMGC:icdcs96.html`.

[FSB+98] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, Joâo Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Roberts, Fadi Sandakly, George Coulouris, Jean Dollimore,

[11] IEZ is an industrial partner involved in the PerDiS project, see `http://www.perdis.esprit.ec.org/members/`.

Paulo Guedes, Daniel Hagimont, and Sacha Krakowiak. PerDiS: design, implementation, and use of a PERsistent DIstributed Store. Technical Report QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98, QMW, CSTB, INRIA and INESC, October 1998. `http://www-sor.inria.fr/publi/PDIUPDS_rr3525.html`.

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

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[PN77] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, June 1977.

[Sal99] Alexandru Salcianu. Extraction et utilisation des informations de type pour le support des objets répartis. Mémoire de dea, DEA d'Informatique de Lyon, INRIA, Rocquencourt (France), July 1999. `http://www-sor.inria.fr/publi/EUITSOR_dea-salcianu-1999-07.html`.

[Ste83] C. J. Stephenson. Fast Fits: New methods for dynamic storage allocation. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 30–32, Bretton Woods, New Hampshire, October 1983.

[WJ98] Paul R. Wilson and Mark S. Johnstone. The memory fragmentation problem: Solved? In *Proc. Int. Symposium on Memory Management (ISMM'98)*, pages 26 – 36, Vancouver, Canada, October 1998. `ftp://ftp.cs.utexas.edu/pub/garbage/malloc/ismm98.ps`.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), September 1995. `ftp://ftp.cs.utexas.edu/pub/garbage/allocscr.ps`.

[ZG92a] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. Technical Report CU-CS-604-92, Dept. of Comp. Sc., Colorado University, Boulder, Colorado (USA), July 1992. `ftp://ftp.cs.colorado.edu/pub/cs/techreports/zorn/CU-CS-604-92.ps.Z`.

[ZG92b] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. Technical Report CU-CS-603-92, Dept. of Comp. Sc., Colorado University, Boulder, Colorado (USA), July 1992. `ftp://ftp.cs.colorado.edu/pub/cs/techreports/zorn/CU-CS-603-92.ps.Z`.