Object-level tracing toolkit: design, implementation, and purpose

Darko Stefanović

September 12, 1995

1 Introduction

The object-level tracing toolkit (OLTT) is a collection of an interface and several optimised algorithms and data structures intended for tracking individual objects in any object-based programming system. A variety of tracing tools can be built using the provided code to gather different object-level statistics.

2 Interface

OLTT defines a simple interface between the instrumented object-based programming system and the object-level tracing routines. The interface makes only rudimentary assumptions about object layout, thus object-level tracing can be used with any foreseeable object-based system.

The only assumptions made about object layout in memory are *that* each object consists of a number of memory words, *that* there is a distinguished first word, *that* the first word at any time has an address, *that* no two objects have the same first word address at any time, and *that* other words in an object are identified by their offset from the first word (which need not be related to the true memory offset).

Under these assumptions, non-contiguous objects are permitted, which is useful for tracing the Smalltalk system. However, for each object field access, the burden of calculating the object first word address and offset is left to the instrumentation. This is reasonable in an interpreted system such as Smalltalk. In a system where instrumentation is achieved by instruction-level tracing, it may not be convenient or possible to find the object header given an object field. However, if the object is known to be contiguous, OLTT can simply find the unique object that contains the field. This is the purpose of the alternative access-noting calls in the interface.

In summary, a tool works in one of two possible modes. 1. Explicit offset mode: each object is created at a distinct address, and each field of the object is (logically) at an offset of some number of words from that address. This offset may be positive, negative, or zero. Since these offsets need not correspond to physical offsets, all manner of noncontiguous objects can be handled. The logical offsets themselves may, for any given object, constitute an arbitrary subset of the integers. The correspondence between the logical offsets and the declared size of the object is not maintained. The computation of offsets must be done on the language side. 2. Implicit offset mode. Each object is physically contiguous, and when an object is created in OLTT, the address of its physically first word must be given, along with its actual size, i.e, number of words. When presented with a physical field address, the tool performs object identification and offset calculations. Computed offsets correspond to physical offsets and fall in the range 0 .. size - 1.

2.1 The interface in detail (signatures in `oltt.h`)

The following are functions to be called by a tracer, or by instrumented code, to track object allocation and deallocation.

- `void oltt_init ()` — Initialise oltt tables; must be called ahead of any other calls to OLTT

- `void oltt_finish ()` — Summarise and print any data tracked by the OLTT. What is tracked and what is printed depends on the tool.

- `void oltt_create_object (void *address, int size, int type, int external_time)` — Record the creation of an object at given address, and given size in words.

  The size argument, which expresses the object size in words, is used for three purposes: 1. If the system has contiguous objects and the instrumented system uses field addresses for calls to OLTT, then the size is used to determine the extent of each object. 2. Certain tools may collect statistics of object size distribution, or collect other statistics separately according to object size. 3. Many tools use object size to calculate the total amount of allocated data, and use this amount as a measure of elapsed time.

  The type argument may be used by some tools to collect separate statistics according to object type. Using the class of an object as its type when tracing a system like Smalltalk, without any adjustements, will produce incorrect results because the apparent type of the object changes whenever its class object is moved during a garbage collection. Therefore, to ensure correctness, a tool used for such a system must register the classes of the tracked objects, and must be able to update the type information.

  The external time argument is used by many tools to compute time-dependent statistics, such as object lifetimes, or inter-access gaps, in those cases where the amount of allocated data is not a good time measure. Which measure of time is appropriate depends on the intended use. If instrumentation is accomplished by instruction-level tracing, then the instruction count is a possible measure.

- `void oltt_release_object (void *address, int external_time)` — Record the deletion or marking for deletion of an object. In garbage-collected systems, a series of these calls is made upon each collection. The address must be the same as given in the call to `oltt_create_object`. Most tools will summarise the information held about the object at this time, so as not to have to keep large amounts of tracking data indefinitely.

- `void oltt_relocate (void *old_address, void *new_address, int external_time)` — Record the relocation (usually by a copying collector) of an object. OLTT does not check the validity of the new address; it is up to the instrumented system to ensure that the new address does not coincide with an existing object address, which in any reasonable system should be the case.

- `void oltt_print_stats ()` — Print statistics of OLTT data structures. Each tool determines the scope and format of the printed statistics.

- `void oltt_print_all_objectdata ()` — Print information about all objects currently tracked. Each tool determines the scope and format of information printed.

The following are functions the instrumented system calls to track accesses to object fields: two kinds of write access (initialising and updating) and read access.

- `void oltt_initialise_at_offset (void *address, int offset, void *value, int external_time)` — Record the initialisation of an object field. The field is at the given `offset` from the given address of the first word of the object. Some tools may make use of the supplied field offset, and some not. Those tools that do use the offset should be able to handle negative offset values. The `value` is the initialising value for the field.

- `void oltt_initialise_contiguous (void *address, void *value, int external_time)` — Record the initialisation of an object field under the assumption that all objects are contiguous. The address given is the memory address of the accessed field; OLTT itself will find the enclosing object.

- `void oltt_update_at_offset (void *address, void *value, int external_time)` — Record the update of an object field (mutation).

- `void oltt_update_contiguous (void *address, void *value, int external_time)` — Record the update of an object field (mutation).

- `void oltt_read_at_offset (void *address, void *value, int external_time)` — Record the read access to an object field.

- `void oltt_read_contiguous (void *address, void *value, int external_time)` — Record the read access to an object field.

Finally, certain tools (in particular, tools that investigate garbage collection strategies) may require the instrumented system to provide some interpretation of object fields. This is the callback interface function to be provided on the language side. N.B. This part of the interface is subject to revision. We have to see how much of the pointer structure is needed for gc algorithm simulation***.

- `extern boolean for_oltt_is_pointer (void *object_address, int allocated_type, int field_offset, void *field_value)` — Determine whether an object field contains a pointer value. OLTT provides the address of the object, the type of the object as given to OLTT at the time of creation, the offset of the field, and the current value of the field (this value is often a pointer, hence the type `void *`).

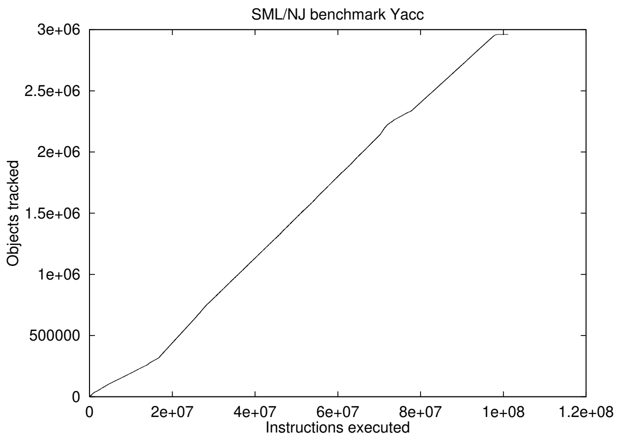
The listed function calls form the OLTT interface, found in the header file `oltt.h`. Each tool is expected to conform to this interface. We have built a number of tools that do, and developed basic object-tracking data structures and associated algorithms. Rather than provide a fixed library of calls, we expect that each tool will be written afresh, but using these common data structures. Tracking tools always require significant processor time and memory resources, and we have found that hand-tuning each tool is important. This fine-tuning consists of determining what the desired statistics are, how they can be computed, what information must be kept during program execution, and what can be safely summarised.

3 Basic implementation

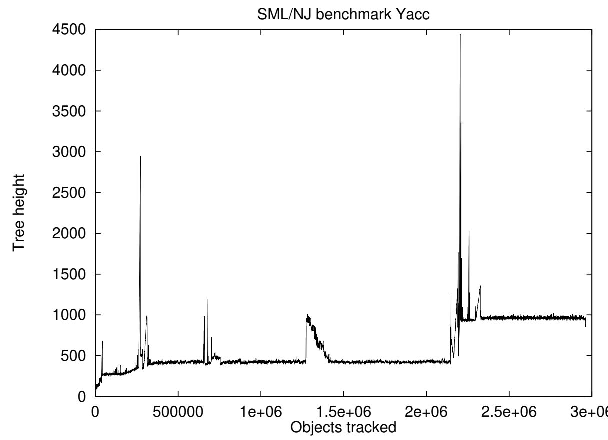
Each tool will differ in the kind and quantity of statistics gathered. The efficiency of tools depends on the implementation of basic object tracking—the task of mapping an object address (in the traced system) to the address of object-related data (in the object-tracking tool). Some earlier systems have used hash tables for this mapping [9]. We find hash tables inadequate for several reasons: 1. Objects are usually allocated sequentially in a contiguous nursery; therefore the addresses that serve as keys into the hash table are sequential. It is difficult to find good fast hashing functions for this case. 2. It is difficult to size hash tables properly, as the number of objects tracked varies widely between programs and also during the execution of a single program. Thus hash tables either waste a lot of space or require rehashing when they are resized. 3. Hash tables allow searching for exact key matches. Since we want to support object identification from field addresses whenever the assumption of object contiguity holds, we prefer a data structure that allows inexact matches.

Instead of hash tables, we use self-adjusting binary search trees (splay trees) [4, 6]. Although the sequential allocation sequence causes the trees to become quite imbalanced, the inherent locality of object accesses is well matched to the self-adjusting operation of splay trees, and the average access time (number of splay operations) is remarkably low. We report below some measurements we obtained using an instrumented version of the splay-tree code. Splay trees have a significant size overhead (each node contains a key, its value, and 3 pointers, thus the overhead is 150%), but only use space proportional to the number of objects tracked. Lastly, splay trees allow inexact matching, so an address can be mapped to the address of the nearest object. A further check based on the declared object size is needed to ensure that the address is indeed *within* the object.

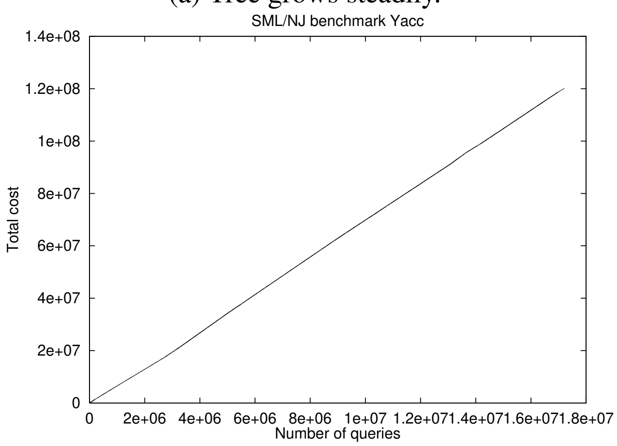
To show how well-suited splay trees are to the task of object mapping, we examine an extreme case. We trace the execution of the SML/NJ benchmark Yacc, without performing any garbage collections (Figure 1). The number of instructions executed is 100 million,



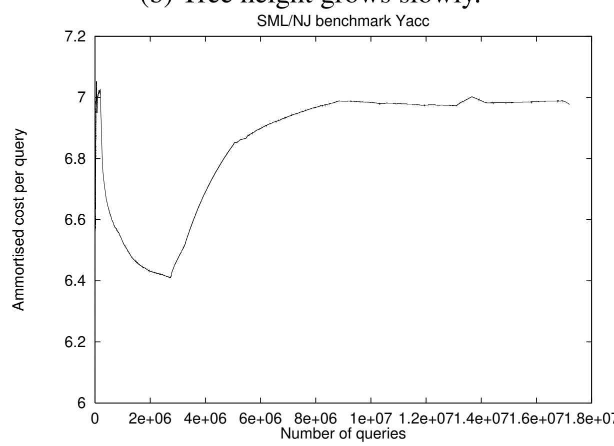
(a) Tree grows steadily.



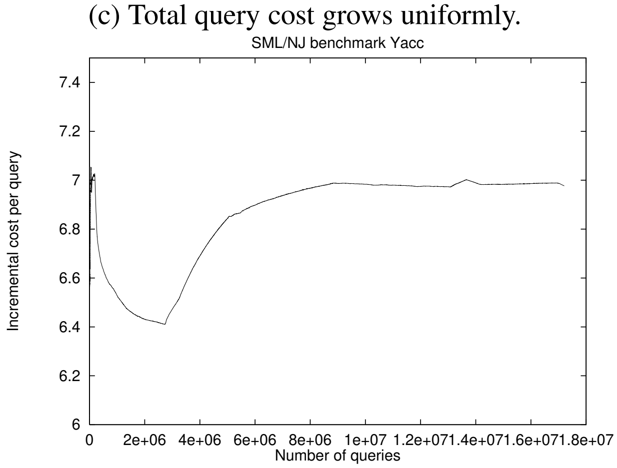
(b) Tree height grows slowly.



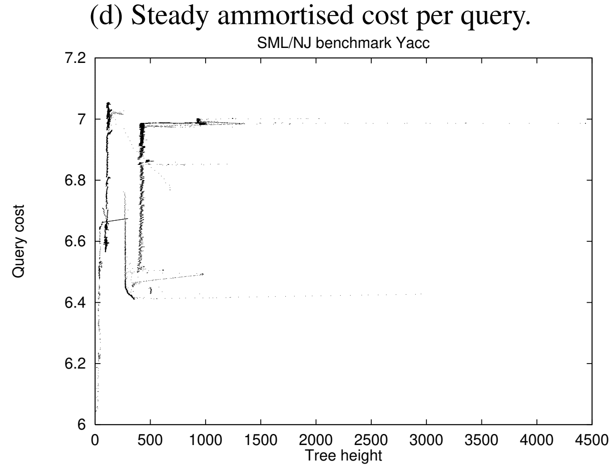
(c) Total query cost grows uniformly.



(d) Steady ammortised cost per query.



(e) Steady incremental cost per query.



(f) Little correlation between tree height and query cost.

Figure 1: Splay tree performance for object mapping.

the total amount of data allocated is 65MB, contained in 3 million objects. Therefore, the number of objects tracked in the map grows steadily from zero to 3 million (Figure 1a). In a perfectly balanced tree, tree height grows as the logarithm of the number of nodes in the tree. In our case, tree height grows in spurts, caused by sequential allocation, such as the spike at the point 250000 in Figure 1b. The spurts are caused by linear object allocation, and the trees are quickly brought into balance by subsequent object accesses. Each query, that is, each call to the splay tree operation `contains_node` is converted into a call to the instrumented version, which computes a cost measure. Each *splay* operation is assessed the cost of one plus the number of link traversals. As shown in Figure 1c-f, in spite of the significant increase in tree height over the course of execution, the query cost measured as the number of splay operations per query remains fairly constant. In fact, the cost varies between 6.4 and 7 and there is little evidence of a correlation between tree height and cost.

Most tools will collect statistics in the form of various distributions, for example, the distribution of object lifetimes. For reasons of convenience, we use splay trees to collect distribution data as well. For many statistics, we have also found it useful not to collect distributions on a linear scale. Indeed, object lifetimes range from a few instructions to hundreds of millions. We keep a logarithmic scale distribution for the entire range, and a linear scale distribution for a low range (often we are especially interested in the frequency of the small values, such as 0).

4 Instrumented systems

We have instrumented two systems to date.

The first system is a Standard ML implementation. We use Standard ML of New Jersey, version 0.93, equipped with an instrumented version of the UMass Garbage Collector Toolkit instead of the original collector. We trace the execution of the compiled ML code using the Shade instruction-level tracer, running on a Sparc 20 workstation. The execution of the garbage collector is not traced. Object allocation instruction sequences are recognised by the tracer and calls to OLTT are issued to mark object creation. Field initialisation is not tracked, as it always accompanies object allocation. Field reads and writes are tracked. The garbage collector produces a relocation map on each collection; it maps the address of each object in the generations scavenged to its new address if the object survived, or to 0, if it is garbage. This relocation map is communicated to the Shade tracer and passed on to OLTT. The external time measure is the number of instructions executed outside of garbage collection.

The second system is the UMass Persistent Smalltalk interpreter. The interpreter has been modified to issue calls to OLTT at appropriate places. Allocation, relocation, and disposal are easily identified; a relocation map is kept by the garbage collector. The Smalltalk interpreter already implements a write barrier; the OLTT call to note object updates is added there. The interpreter does not have a read barrier, but it does check object residency for the purposes of persistence. The OLTT calls to note object reads are placed here and in the bytecode realisations that require it. It should be noted that in an interpreted system there is some latitude in deciding which accesses to objects are to be traced: some result from bytecode interpretation, and some result from bookkeeping in the interpreter. The external time measure is the number of bytecodes executed.

5 Performance

In this section we present results of measuring the execution speed of instrumented programs and tracking tools. Since instrumented programs typically run more slowly than original programs, and have greater memory demands, it is important to know how great this slowdown (dilation) and memory expansion are.

Since there are several sources of tracing overhead we present the measured execution times for several degrees of tracing support:

1. Uninstrumented program.
2. Instrumented, untraced program: in systems with garbage collection, it is often necessary to add instrumentation to the garbage collector, to record object relocation and deallocation. As a result, the garbage collector is slowed down,[1] even without tracing—for example, the entire old space must be scanned to identify dead objects, even if the collector is a copying one.[2]
3. Program with instruction counting using the standard Shade tool `icount`.
4. Program with custom instruction-level tracing and OLTT calls to a *no-op* tool (each function returns immediately).
5. Program with tracing and OLTT calls to a *bare object tool* (objects are tracked, but no statistics are taken).
6. Program with tracing and OLTT calls to a *bare access tool* (objects and field accesses are tracked, but no statistics are taken).
7. Program with tracing and OLTT calls to a tool that collects statistics on object lifetimes and inter-access gaps (this tool keeps access records during execution).
8. Program with tracing and OLTT calls to a *bare tool* based on hash tables.

Measurements of object tracking costs are presented in Table 1. It will be seen in the table that the overheads of object-level tracking are significant, but not prohibitive. In particular, they can be well afforded in systems that are already slowed down by instruction-level tracing. It is also clear that overheads depend very much on the properties of the traced program, such as the average number of objects tracked and the average number of accesses per object. This dependence was to be expected.

[1] Once the garbage collector has been instrumented, its behaviour is changed, and thus there is no point in tracing *it*. In our instrumentation of the ML system, we instruct Shade not to trace instructions during garbage collection.

[2] For an overview of garbage collection algorithms, see Ref. [8].

Benchmark	SML: ML	Smalltalk: interactive	Smalltalk: calculus
uninstrumented	12	9	55
instrumented, untraced	44		
`icount`-traced	303		
noop	3801	17	107
bare obj	4046 (+6.4%)	20 (+17.6%)	131 (+22.4%)
bare acc	4422 (+16.3%)	49 (+188.2%)	219 (+104.7%)
life & gap stats	4984 (+31.1%)	89 (+423.5%)	302 (+182.2%)
bare obj hash	3843 (+1.1%)	47 (+176.5%)	176 (+64.5%)
total objects	7 811 072	126 976	499 712
avg. num objects	126 632.17	885.40	6346.42
avg. splay ops	9.85	10.86	11.48
avg. hash ops	4.07	779.02	574.50

Table 1: Object tracking costs

Finally, we compare object mapping using splay trees and using hash tables. The comparison is necessarily limited to one particular implementation of each method. This should be borne in mind especially for hash tables, since we did not explore to any great extent the effects of varying the hashing function.

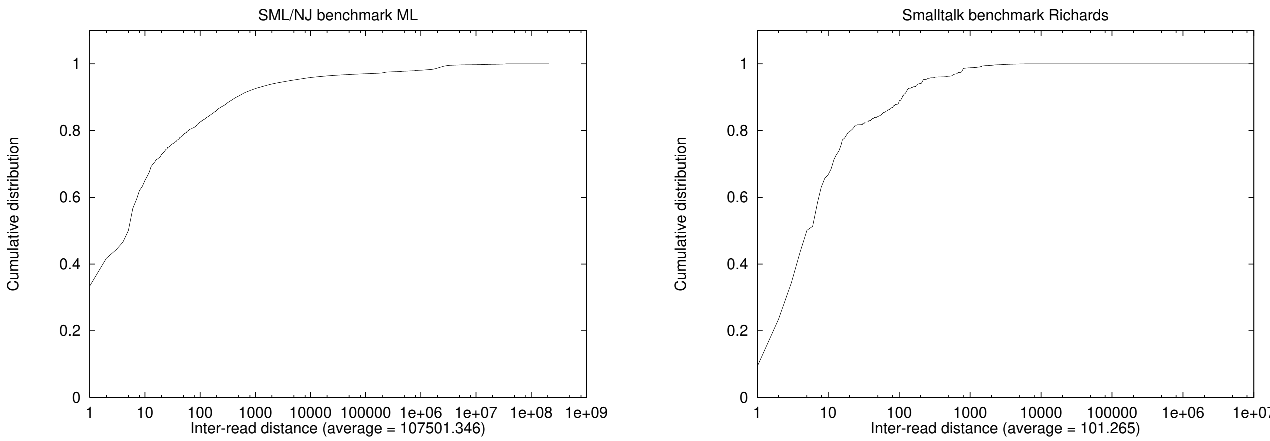
The comparison is also limited to the bare object-tracking tool. The table shows that the overhead of hashing is 6 times smaller for the ML benchmark, but 10 times greater for the Smalltalk benchmarks. The cost rows of th etable show that hashing requires looking up a very large number of hash table entries for the Smalltalk benchmarks, but a reasonably small number for ML. Furthermore, the expense of a hash table lookup is lower than that of a splay operation. It will also be noted that splay tree costs vary much less across the traced programs than hash table costs.

It might appear that hashing should be preferred for tracking objects in ML; unfortunately, the tool implemented using a hash table is not capable of mid-object field identification, which the construction of the ML instruction-level tracer necessitates.

6 Example tool

We built an OLTT tool to find the distribution of object inter-access gaps. An inter-read gap is the gap between two successive reads of an object, or the gap between an update of the object and the first following read. An inter-update gap is the gap between two successive updates, or the gap between object initialisation and the first update. An on-line statistical analysis of inter-access gaps may be used to predict future accesses and improve cache replacement policies (the decision which cached entries to eject to make room for new entries); the resulting policies are consistently better than least-recently-used replacement [3].

For our Standard ML system, executing benchmark ML [5], the ML compiler, we obtained the distribution in Figure 2a. For our Smalltalk system, doing the built-in macrobenchmark interactive [1], we obtained the distribution in Figure 2b.



(a) SML/NJ. (b) Smalltalk.

Figure 2: Cumulative distribution of object inter-access gaps.

7 Support for object management simulation

8 Related work

Ungar and Jackson reported object lifetime statistics for objects surviving a certain time [7]. Zorn devised an object tracing tool called MARS for use in the study of garbage collection performance [9]. MARS is somewhat similar to OLTT; it is linked to a Lisp environment and shadows objects using a hash table mapping.

An interesting parallel is that with hardware mechanisms to assist real-time garbage collection as described by Nilsen and Schmidt [2]. There the lookup operation, which determines the object header location given a pointer to any location within the object, is considered the most frequent and hardware support assures that it executes in fixed time.

References

[1] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992. *ACM SIGPLAN Notices 27*, 10 (October 1992).

[2] Kelvin D. Nilsen and William J. Schmidt. Cost-effective object space management for hardware-assisted real-time garbage collection. *ACM Letters on Programming Languages and Systems*, 1(4):338–354, December 1992.

[3] Vidyadhar Phalke and Bhaskarpillai Gopinath. An inter-reference gap model for temporal locality in porgram behavior. In *1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Ottawa, Canada, May 1995*, pages 291–300. ACM, 1995.

[4] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. In *Proceedings of the ACM SIGACT Symposium on Theory*, pages 235–245, Boston, Massachusetts, April 1983.

[5] Darko Stefanović and J. Eliot B. Moss. Characterisation of object behaviour in Standard ML of New Jersey. In *1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994.

[6] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

[7] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–17, San Diego, California, September 1988. *ACM SIGPLAN Notices 23*, 11 (November 1988).

[8] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag. Extended version, submitted to ACM Computing Surveys.

[9] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, December 1989. Available as Technical Report UCB/CSD 89/544.