The 1991 Workshop on Garbage Collection in Object-Oriented Systems

Paul R. Wilson*
University of Texas at Austin

Barry Hayes†
Stanford University and Xerox PARC

...we have the ability to handle garbage more intelligently, using both emerging technologies and new and improved versions of older solutions...
—Mariette DiChristina, "How we can win the war against garbage," *Popular Science*, October 1990, p. 57.

1 Overview

This workshop consisted primarily of 10-minute presentations by researchers and software developers, interspersed with a few 20-minute presentations and a panel discussion on conservative vs. accurate approaches to garbage collection.

While the format was partly due to our indecisiveness in selecting the "best" papers for presentation, it also reflected the generally good quality of accepted papers and the high standard of background knowledge among the participants. We got several comments to the effect that the 10-minute slots were "just right," because nearly all of the attendees had sufficient background that little preparatory explanation was needed—speakers could quickly get to the novel and interesting points of their papers.[1] Future workshop organizers might want to take the success of this format into account.

Given the pleasant result of this self-organizing anarchy, we present the workshop report in much the same spirit—we are mostly presenting the abstracts of the accepted papers and panelists' positions. The ordering below is roughly the one in which papers were presented. (A few papers weren't actually presented, but we include their abstracts here.)

Full versions of most of the presented papers are available in PostScript form via anonymous internet ftp from cs.utexas.edu, in the directory pub/garbage/GC91. Papers from the first workshop (organized by Eric Jul and Niels Christian Juul) are in pub/garbage/GC90. We are also compiling a bibliography which will be made available there.[2]

We would like to thank all of the participants in the workshop, including our special invited guests David Chase (of Sun) and Andrew Koenig (of AT&T), and our panelists Joel Bartlett, Hans Boehm, and Eliot Moss.

2 Morning Session Abstracts

On Multi-Threaded List Processing and Garbage Collection

Wolfgang Kuechlin and Nicholas Nevin, Dept of Computer and Information Science, The Ohio State University, Columbus, Ohio 43210-1277, U.S.A. {keuchlin,nevin}@cis.ohio-state.edu.

We discuss the problem of parallel list-processing based on *threads of control*, such as Mach's C Threads. Our main insight is that the threads paradigm and the use of a standard threads system suggests a heap memory layout and garbage collection technique which is quite different from existing Lisp and Prolog Systems.

A High-Performance Architecture for Real-Time Garbage Collection

Kelvin Nilsen, Dept. of Computer Science, Iowa State University, Ames, IA 50011 (515) 294-2259 (kelvin@cs.iastate.edu uunet!atanasoff!kelvin)

Hardware-assisted garbage collection offers the potential of high average-case allocation rates and mem-

*Paul R. Wilson, Dept. of Computer Sciences, University of Texas, Taylor Hall 2.124, Austin, Texas 78712-1188 (internet: wilson@cs.utexas.edu).

†Barry Hayes, Xerox PARC, Room 2113 CSL, 3333 Coyote Hill Road, Palo Alto, California 94304 (internet: bhayes@neon.stanford.edu).

[1] We also adjusted the ordering of the talks to present related topics together, and many of the presenters conferred to make sure their presentations were not too redundant. In fact, some people thought this made the few 20-minute slots a shade too long.

[2] A similar repository, maintained by Niels Christian Juul, is on ftp.diku.dk for European access.

ory bandwidth, with very low worst-case allocation, fetch, and store times. This paper describes an architecture that allows memory fetch and store operations to execute, on the average, nearly as fast as traditional memory. The architecture is high-performance in that it includes support for caching of garbage-collected memory cells. The architecture is real-time in that the worst-case time required for a memory fetch or store is approximately six traditional memory cycles, and the time required to allocate an object is bounded by a small constant times the size of the object. A prototype of the proposed architecture has been successfully simulated. Continuing research focuses on measuring the system's performance under real workloads.

How Real is "Real-Time" GC?

P. T. Withington, Symbolics, Inc., Eight New England Executive Park, East Burlington, MA 01803, U.S.A. (ptw@Jasper.SCRC.Symbolics.COM)

A group at Symbolics is developing a Lisp run-time kernel, derived from its Genera(R) operating system, to support real-time control applications. The first candidate application has strict response-time requirements (so strict that it does not permit the use of paged virtual memory). Traditionally, Lisp's automatic storage-management mechanism has made it unsuitable to real-time systems of this nature. A number of garbage collector designs and implementations exist (including the Genera garbage collector) that purport to be "real-time", but which actually have only mitigated the impact of garbage collection sufficiently that it usually goes unnoticed by humans. Unfortunately, electro-mechanical systems are not so forgiving. This paper examines the limitations of existing real-time garbage collectors and describes the avenues that we are exploring in our work to develop a CLOS-based garbage collector that can meet the real-time requirements of real real-time systems.

Automatic Storage Management for Systems with Real Time Constraints

Stephen Engelstad (marv1@iexist.att.com) and James E. Vandendorpe, AT&T Bell Laboratories, Naperville, IL 60566

A garbage collector programmed in C reclaims objects by executing during the allocation of new objects. After a dynamically scheduled number of object allocations the application is delayed while lost resources are recovered. The average delay is 6 ms on a Sun3 and 2 ms on a Sun4 with a worst case delay of 10 ms on both. To achieve these results we refined the generational algorithm published by Carl Hewitt and Henry Lieberman. Our algorithm is implemented within the Calico programming system.

Comparing Two Garbage Collectors

Douglas Johnson, Texas Instruments MS 369, P.O. Box 655621, Dallas, Texas 75265 (johnson@ti.com)

Based on an experiment comparing two garbage collection algorithms on a large, long-running Lisp program, we find that a simple, two generation stop-and-copy collector is equal or superior to a complex, multi-generational incremental collector in every measured area except paging performance in limited physical memory sizes. The major conclusion from this experiment is that it is difficult to justify the cost of complex incremental collectors unless they are used to optimize paging performance via object reordering. However, there are interesting classes of programs that can get significant benefit from such optimization.

Simple GC-Safe Compilation

Hans-Juergen Boehm, Xerox PARC, 3333 Coyote Hill Road, Palo Alto, California 94304, USA (boehm@xerox.com)

We propose that C compilers (and translators from future standard intermediate languages) optionally produce code that guarantees safety in the presence of a conservative garbage collector. It is argued that this is essential, and that it should have negligible impact on object code performance. It is easy to specify what is required of the compiler. The necessary changes to the compiler are independent of the original source language, be it C itself, or another language that is compiled to C.

The UMass Language Independent Garbage Collector Toolkit

J. Eliot B. Moss, Department of Computer Science, University of Massachusetts, Amherst, MA 01003 (moss@cs.umass.edu)

At the OOPSLA/ECOOP '90 workshop on garbage collection Rick Hudson and Amer Diwan presented a scheme for flexible scavenging based on a varying number of fixed size generations. We have since developed a new scheme that we believe is even more flexible. The sizes of generations can vary, as well as the number of generations, and we provide very efficient age-based promotion using no age counters or high water mark tests. In addition, we have implemented the scheme in such a way as to separate language specific parts (e.g., details of locating roots for a scavenge) from language independent components. The result is a toolkit that substantially reduces the effort in supporting scavenging in new language implementations.

Stack Tracing In a Statically Typed Language

Amer Diwan, Object Oriented Systems Laboratory, Department of Computer Science, University of Massachusetts, Amherst, MA 01003, (413) 545-0256 (diwan@cs.umass.edu)

Statically-typed languages such as Modula-3 provide many opportunities to mitigate the costs of garbage collection. Compiler generated tables can assist the collector in locating the root pointers and in updating them when objects are moved. This obviates the need for run-time tags in the stack and offers substantial performance improvements.

In this paper, we describe some key issues that need to be addressed before we have an efficient stack tracing scheme. We also describe our solutions to these problems in the context of Modula-3.

Main Memory Management for Persistence

Antony Hosking, Object Oriented Systems Laboratory, Department of Computer and Information Science, University of Massachusetts, Amherst MA 01003 (hosking@cs.umass.edu)

Reachability-based persistence imposes new requirements for main memory management in general, and garbage collection in particular. After a brief introduction to the characteristics and requirements of reachability-based persistence, we present the design of a run-time storage manager for Persistent Smalltalk and Persistent Modula-3, which allows the reclamation of storage from both temporary objects and buffered persistent objects.

Finalization in a Garbage Collected World

Richard L. Hudson, Research Associate, University Computing Services, University of Massachusetts at Amherst (hudson@cs.umass.edu)

With the advent of object oriented technologies, data structures have become more complex and the tradition of relying on programmers to explicitly deallocate objects has become tenuous. To relieve the programmer of the increasingly intractable problem of detecting references that have expired, modern run-time platforms use garbage collection to help manage memory. However, before an object can be deallocated, it is sometimes necessary to execute a *finalization* routine associated with the object. Such a routine might include closing an open file or informing the windowing system that a window is no longer needed. In some languages, the only way to free storage is to execute finalization routines that explicitly deallocate the object. On the other hand, garbage collection algorithms automatically detect when objects are no longer needed. This frees the user from having to deallocate storage explicitly in a finalization routine, but does not address the problem of executing finalization routines that have other responsibilities. In this paper, we propose semantics to extend finalization to include garbage collected objects. In the context of a type accurate generational garbage collector, we describe an implementation strategy that allows finalization to be performed concurrently with the user's threads.

Outwitting GC Devils: A Hybrid Incremental Garbage Collector

David Ungar and Frank Jackson, ParcPlace Systems, Inc., 1550 Plymouth St., Mountain View, CA 94043 (David.Ungar@sun.com, Jackson@parcplace.com)

No garbage collector is perfect, and it is possible to invest a great deal of effort in a particular design of a collector before the extent of its weaknesses become apparent. Even large benchmarks may fail to expose a flaw that can interfere with attaining acceptable performance for other application programs. Designers need to build collectors that exhibit robust behavior in the face of a wide variety of application program allocation and reclamation patterns. In order to focus on this goal, it helps to imagine an adversary who writes application programs with the specific purpose of confounding the garbage collector. In fact, in developing a collector for a commercial product, we found that this was more than a conceptual device; users did (unwittingly) construct some programs that exhibited peculiar behavior which caused an early version of our collector to fail. As a result, we implemented a hybrid collector combining Generation Scavenging, with an incremental mark-sweep algorithm. Its design was influenced by the desire to avoid potential pathologies. Users report satisfaction with the new collector. We are waiting to see what its pathologies may be.

Garbage-Collection Bugs That I Have Known

Frank Jackson, ParcPlace Systems, 1550 Plymouth Street, Mountain View, CA 94043 (415) 691-6710 (jackson@parcplace.com)

This paper takes a light-hearted look at some of the more unusual bugs that have bedeviled the three generations of automatic memory managers for the Smalltalk-80 system. These bugs are worth revisiting for the following reasons:

1. Because it can be instructive to examine the kinds of problems that are uncovered when a memory manager that seems to work perfectly well in the lab is actually put to the test by real users running real applications; and

2. Because such bugs often occur in precisely those areas where the technology is evolving at the most rapid pace.

The specific areas of automatic memory management that are examined in this paper include: weak pointers, finalization, hybrid memory managers, and adaptive storage-reclamation algorithms. Those bugs that are discussed include: the Reincarnation Bug, the Resurrection Bug, the problem of Premature Finalization, and the Pig-in-a-Python phenomenon.

The Myth of High Object Allocation Rates

Urs Hoelzle, Computer Systems Laboratory, CIS 57, Stanford University, Stanford CA 94305 (urs@self.stanford.edu)

Garbage collector designs are often motivated by the high allocation rates displayed by the system for which the collector is intended; this furious object creation rate is accepted as a given.

I argue that a promising way to reduce the GC overhead is to reduce the amount of garbage produced, i.e. to lower the allocation rate of the system. “Garbage prevention” should be an integral part of any system using garbage collection, and the potential gains of compiler optimizations aimed at reducing the allocation rate could well outweigh the gains achievable by just tuning the garbage collector. Some measurements of the Self system are used to illustrate this point.

Cost of Garbage Collection in the Self System

Craig Chambers, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195 (chambers@cs.washington.edu)

Differences in the overhead of garbage collection for different systems stem from differences in the language implementation technology and in typical programming style. Ungar reports remarkably low overheads of 2-3% for Generation Scavenging in Berkeley Smalltalk and SOAR, but these Smalltalk systems had relatively unoptimized language implementations relative to traditional languages and fostered high object allocation and death rates relative to traditional languages.

Self provides a new data point in this matrix: an optimizing language implementation combined with high object allocation and death rates. Optimizations in the Self system both dramatically speed up normal program execution compared to Smalltalk systems and reduce the object allocation and death rates by optimizing away many closure creations. We measured the overhead of garbage collection in Self on four large benchmarks and found overheads ranging from 4% to 27%. Interestingly, store checking consumes a major fraction of this overhead.

3 Panel Discussion: Conservative vs. Accurate Garbage Collection

Hans Boehm

By conservative garbage collection we mean garbage collection in the presence of values for which it is not known whether or not they are pointers. There are various degrees of conservative garbage collection depending on how much pointer information is known. For example, the layout of heap allocated objects may be known, while stack layout may not be known.

Garbage collectors that accommodate a high degree of conservatism, i.e. that can operate with minimal pointer location information are currently desirable for two reasons. First, new code that relies on garbage collection should interoperate correctly with traditional C code. Thus the garbage collector must be able to understand references from C data structures. Second, new programming language implementations increasingly compile to C to obtain both portability and access to machine dependent optimizers that were built with an (often proprietary) understanding of hardware timing. This makes it difficult or impossible to provide data structure layout information to the collector.

Even very conservative, nonmoving garbage collectors can provide performance that is often better than malloc/free performance, and competitive with other kinds of collectors. They may, on rare occasions, retain noticeably more unreachable data than other types of collectors. But essentially no existing garbage collector provides guarantees on what storage will be reclaimed. Conservative garbage collectors impose some minimal requirements on code optimizers, but these can be easily standardized, and are independent of the original source language.

Joel Bartlett

The terms "accurate" and "conservative" are an attempt to put black and white labels on a gray scale that measures the amount of information that the garbage collector has to work with. People don't build conservative collectors because they're lazy, sloppy, or have loose morals. They build them because they wish to provide garbage collection in inhospitable environments. When faced with the choice of not enough information at runtime, or a "GC unsafe" language, one can abandon the implementation, or use conservative techniques. As I'm interested to bringing garbage collection to more users, I do the later.

At WRL, we have used "mostly-copying" garbage collection to build an efficient, portable Scheme-to-C compiler, Scheme->C, and over 40,000 lines of CAD tools in C++. In spite of the fact that it's easy to demonstrate how conservative techniques can cause memory leaks, our experience is that this is a rare occurrence in production programs. Challenges for the future include convincing others that conservative collection is preferable to bare-handed allocation and that code generators should preserve the minimal invariants required by conservative collectors.

Eliot Moss

In order to understand my position on garbage collection strategies, it is useful to see my taxonomy of the strategies. There are *ambiguous roots* collectors (frequently called *conservative* because they must *assume* a quantity is a pointer unless proven otherwise) and *unambiguous* ones. Ambiguous roots collectors are simple in the absence of certain compiler optimizations, but making language implementation *gc-safe* requires additional analysis in the compiler, and will sometimes affect the object code (e.g., to retain base pointers that would otherwise be considered dead). Unambiguous roots collectors need at least to know the types of most things, hence they are predicated on *type accuracy*. For good performance, one can avoid tagging pointers at run-time, at the expense of additional compiler work and space consumed by tables. The compiler's analysis can be increasingly refined, striving towards perfect *precision* (knowing exactly what is garbage and what is not), which is undecidable.

My main point is that once you go to the effort of enhancing a compiler to make its output gc-safe, you might as well go the extra mile of supporting type accuracy. The improved accuracy avoids the accidental retention of space that sometimes happens with ambiguous roots collectors, but more importantly, type accuracy allows objects to move, and thus allows a wider range of gc strategies, especially compaction. This is important in our work on persistent programming languages, but my position is that it is also important in the absence of persistence. Thus, I hope that "conservative" collectors will only be a stop-gap until better techniques are deployed. (Note: we have indeed developed the necessary compiler technology for our Modula-3 implementation.)

In my view, these are the *issues*, whether or not you agree with my position:

- Accuracy: How much garbage does the collector fail to reclaim?
- Optimization: Which compiler optimizations may cause the collector to fail?
- Ambiguity: Can objects move? Can the collector and run-time distinguish pointers from non-pointers?
- Effort: How much implementation effort is required?

The only issue on which type accurate collectors are not strong is implementation effort, and we argued above that the effort required is only a little more than that needed for gc-safety, so type accuracy should be our goal.

David Chase

There is an interaction between optimizing compilers and garbage collectors—good optimizers can do an arbitrarily amazing job of hiding pointers (stored in registers, not in the heap) from the garbage collector. There is no part of the C or Fortran or C++ specifications that tells compilers not to do this, and market pressure encourages compilers to do this. The RS/6000 compilers are known to do this today, and I observed Sun's f77 compiler into this in 1988. Do understand that these are not "interior" pointers—these are typically encodings of the form "p - K" (to which K is added to reconstitute the "p") or "p2 - p1" (to which "p1" is added to reconstitute "p2").

Bartlett's conservative collector goes a long way in solving this problem, but to be sure, some optimizer support is also required. For languages that interoperate with C, one must also beware of routines in the C library that return "derived" (interior) pointers based on their parameters.

On another note, I think that "prefetch" instructions for some forms of cache management (see the i860 pipelined load instructions, or recent papers in ASPLOS) may have interesting uses in garbage collectors, and should be investigated. One reason this is a good hardware feature to exploit is that it is useful for Fortran, and thus will probably become widely available (as opposed to hardware features designed to support less-popular languages).

4 Afternoon Session Abstracts

Standardizing Memory Management Descriptions

Alan M. Durham and Ralph Johnson, Department of Computer Science, University of Illinois, Urbana-Champaign (durham@cs.uiuc.edu)

This position paper describes my research on describing memory management to an optimizing compiler. The goal is to make optimizing compilers for languages like Smalltalk easier to retarget to new memory management systems.

The Treadmill: Real-Time Garbage Collection Without Motion Sickness

Henry G. Baker, Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436 (818) 501-4956, (818) 986-1360 FAX.

A simple real-time garbage collection algorithm is presented which does not copy, thereby avoiding some of the problems caused by the asynchronous motion of objects. This in-place "treadmill" garbage collection scheme has approximately the same complexity as other non-moving garbage collectors, thus making it usable in a high-level language implementation where some pointers cannot be traced. The treadmill is currently being used in a Lisp system built in Ada.

Garbage Collecting an Object-Oriented Operating System Kernel

Vincent F. Russo, Department of Computer Science, Purdue University, West Lafayette, IN 47907 (russo@cs.purdue.edu)

As part of the *Renaissance* project at Purdue University, we are investigating the feasibility of garbage collecting an actively running operating system kernel. In this paper, we discuss the algorithm we have adapted to our needs, along with the techniques we use to support garbage collection in C++ (the object-oriented language in which we have coded our system).

Towards User (Application) Language-Level Garbage Collection in Object-Oriented Concurrent Languages

Masahiro Yasugi and Akinori Yonezawa, Department of Information Science, University of Tokyo (yonezawa@is.s.u-tokyo.ac.jp)

We are implementing object-oriented concurrent languages on highly-parallel computers without shared memory.Our approach to GC is to translate a user program (which is written in an object-oriented concurrent language) into the user program with GC which is also written in the object-oriented concurrent language. In this approach, the runtime routines for GC is not required when we implement language systems. We will present an example of GC schemes aimed at user-language-level general GC. To illustrate the example, we will also give a general definition of garbage in terms of groups.

Parallel Conservative Garbage Collection with Fast Object Allocation

Shinichi Furusou, Satoshi Matsuoka, and Akinori Yonezawa, Department of Information Science, The University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, Tokyo, 113 JAPAN; Phone +81-3-3812-2111 ext. 4108 (furuso@is.s.u-tokyo.ac.jp, matsu@is.s.u-tokyo.ac.jp, yonezawa@is.s.u-tokyo.ac.jp)

Conservative Garbage Collection (GC) algorithm provides garbage collection independent of programming languages and applications, but is difficult to extend naively into a concurrent algorithm, where allocation speed of up to one million objects per second per thread is required. Our parallel conservative GC is a parallel and real-time conservative GC algorithm, whose allocation and collection can be done almost totally in parallel without the need for synchronization or mutual exclusion. Results derived from proof of correctness of the algorithm allows object allocation to be done totally distributed and parallel, so that the allocation cost is estimated to be as little as 13–15 instructions per object allocation, including the overhead of (rare) synchronization. The preliminary version of our algorithm is already running on a 4-processor LUNA-88K with Mach 2.5. Our algorithm relies on the availability of virtual memory primitives to the user in order to maintain the necessary invariants of the collector.

A System Model of Memory Management

Robert MacLachlan, Department of Computer Science, Carnegie Mellon University
(Rob.MacLachlan@cs.cmu.edu)

As both memory systems and memory managers become more complex, it is increasingly important to develop a unified framework for examining the interaction between memory system hardware and memory management software. This paper explores one possible connection.

Garbage Collection in C++

Paulo Ferreira, INESC/IST, R. Alves Redol n^o 9, 6^o, 1000 Lisboa, PORTUGAL; Phone: (351) 1.3155150, fax: (351) 1.525843, (pjpf@sabrina.inesc.pt)

This paper describes the problems and solutions found when developing a library providing garbage collection for C++ programs. Not only objects are reclaimed, but also non-object memory which is usually allocated with the `malloc()` call. In order to reclaim non-object memory, an incremental mark-and-sweep algorithm is provided. The reclamation of objects can be done by the same incremental mark and sweep, or a multigenerational copy algorithm which drastically reduces the time of a global collection. Pointers to the interior of objects are fully supported. The programmer only has to use a few simple macros in order to have garbage collection in his program. He can also specify which objects are to be garbage-collected on a class-by-class basis. Thus, within an application there might coexist objects that are garbage collected along with others that are not. A preprocessor is being developed to release the programmer from using these macros.

Generational Garbage Collection for C-Based Object-Oriented Languages

Kazushi Kuse and Tsutomu Kamimura, IBM Japan, Tokyo Research Laboratory, 5-19, Sanban-cho, Choyoda-ku, Tokyo

The approach we have taken here is to separate memory management of objects from that of standard C data. We provide automatic reclamation of objects, but do not reclaim C data. For this purpose, we impose restrictions on the use of objects, so garbage collection of objects can be performed safely. Such restrictions are basically to make typing of objects strong, to make run-time type information available, to disallow a network of pointers in the heap as well as the stack, and to disallow pointers to the middle of objects.

DGD: Doom, Gloom and Despair—Distributed Garbage Detection made depressing

Peter Dickman, Cambridge University and ERCIM; currently at INRIA Rocquencourt, BP105, 78153 Le Chesnay Cedex, France
(Peter.Dickman@cl.cam.ac.uk)

The talk is aimed at those who haven't worked with distributed systems, and will introduce some of the main problems that arise in Distributed Garbage Detection. A variety of techniques will be presented, together with the problems that they are intended to solve and those that defeat them. This survey will motivate the use of hybrid mechanisms, for example the one that David will present immediately afterwards. Variants on reference counting, the problems with tracing techniques and the use of non-conservative aging and refresh schemes will all be discussed.

The various ways in which DGD's can be analysed will be mentioned, in particular the various issues in counting messages and analysing space requirements will be presented. Finally, scaling is introduced, with some guidelines on what to look for in a scalable garbage detector.

Distributed Garbage Collector as an Operating System Component

David Plainfosse' and Marc Shapiro, INRIA, BP105, 78153 Rocquencourt Cedex, France (dp@sor.inria.fr)

Recent development of object-oriented technology has sparked interest in low-level support systems for user-defined objects. A number of operating systems and database systems offer such support. Until recently, garbage collection has often been judged to be too language- dependent, too complex, and too costly for general-purpose systems. In contrast, we think that operating systems should be designed and implemented to offer support for garbage collection. Our approach is to provide a generic service for distributed garbage collection, building upon existing, language-dependent, local garbage collectors. We have designed an efficient fault-tolerant distributed garbage collection protocol, and we are currently planning to integrate it in the Soul nucleus, a general-purpose object-oriented system layered above the Chorus micro-kernel. In the paper, we focus on the protocol itself and especially on distribution and reliability aspects.

Local and Global Distributed Garbage Collection

Stephen C. Crawley, Defence Science and Technology Organisation, PO Box 1500, Salisbury 5108, Australia (sxc@itd.dsto.oz.au)

This paper describes three garbage collection algorithms in a distributed object system. The local algorithm collects garbage on a single machine independent of others. The extended local algorithm runs asynchronously on each machine collecting local and distributed garbage. The global algorithm provides faster collection of all distributed garbage including cycles. All of the algorithms will work when parts of the object system are unavailable.