Appeared in *Proceedings of the XIII International Conference of the Chilean Computer Science Society*, La Serena, Chile, Oct. 1993, pp. 251-259.

GARBAGE COLLECTION IN DISTRIBUTED EZ

Alvaro E. Campos[1] and David R. Hanson[2]

[1]Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
Casilla 306, Santiago 22, Chile
`acampos@ing.puc.cl`

[2]Department of Computer Science
Princeton University
Princeton, NJ 08544, USA
`drh@cs.princeton.edu`

INTRODUCTION

EZ [8, 10] is a programming environment that most closely resembles those for APL, LISP, Smalltalk, and related systems. *EZ* integrates the facilities provided separately by traditional programming languages and operating systems by casting traditional operating-system services as language features. The result is a complete computing environment that is intended to be an alternative to both conventional languages and operating systems, and that also supports manipulating the environment.

Distributed *EZ* [6] adapts the recent implementations of shared virtual memory [14] to distribute *EZ*'s virtual address space across a network of work-stations. Identical copies of a distributed memory manager run at each processor, and the entire, persistent, virtual address space is distributed among their secondary storage devices. *EZ* is an interpreted system, and the interpreters are the only clients of the managers, which hide all distribution details from the rest of the system and from the users' programs. Each manager caches the pages that its clients are using in an attempt to collect pages at the processor that accesses them; managers replicate pages to allow multiple readers, but permit only one writer in order to maintain coherence. Each page is owned by the last manager that granted write access to the page to a local interpreter; ownership migrates with write accesses. The pages themselves migrate among the secondary storage devices of the individual processors because the owner keeps the in-disk copy of a page up to date.

Earlier versions of *EZ* used an off-line garbage collector to reclaim inaccessible pages in the disk representation of the virtual address space. This approach worked fine for a prototype and, as shown by the Oberon system [18], is perhaps adequate for a non-distributed, single-user system that is subject to frequent idle periods. But an off-line approach is unsuitable for a distributed system.

Distributed *EZ* uses a distributed, mark-and-sweep garbage collector, which has the usual two phases. First, the *mark* phase marks all virtual-memory pages that are accessible from a few system root objects. Then, the *sweep* phase sweeps the virtual memory and collects all those pages that remain unmarked after the mark phase. *EZ*'s collector is *concurrent*; it works in parallel with the mutators, which are interpreter threads acting on behalf of *EZ* processes. It is also *real time*; the mutators are never interrupted for longer than a small constant time.

For systems like *EZ*, which have a large, persistent address space, efficiency of the collection algorithm is less important than concurrency. Space efficiency and minimal disruption of mutators are more important than time efficiency of the algorithm itself. Indeed, all that is required is that the collector replenish the supply of free virtual-memory pages fast enough so that applications rarely have to wait to allocate a new page; and, also, that it collects all inaccessible pages eventually.

DISTRIBUTED GARBAGE COLLECTION

Most garbage collectors proposed for distributed systems require double indirection for remote references [12, 16, 17]. This mechanism requires direct mutator collaboration, and imposes a severe overhead on the mutators when creating references and, especially, when dereferencing. Some collectors are designed for systems in which pages do not migrate [12, 17], or in which page migration requires mutator collaboration to update some of the collector's data structures [4]. Many proposals require global synchronization among processors [2, 3, 4, 9, 13], which in some cases implies halting the mutators. Finally, in some parallel or distributed systems collections occur in a per-process basis with no global garbage collection [11], or they have been serialized disallowing concurrency [15].

Collectors based on reference counting are an interesting alternative because they are relatively simple and incremental. They do, however, require extra storage per reference, and have two major weaknesses. First, maintaining the counts requires direct collaboration from the mutators whenever they copy or remove a reference. Second, and more important, these algorithms cannot handle cyclical references, which makes them unsuitable for *EZ*, where cycles abound.

Copying collectors have the potential to be much more efficient than reference-counting and mark-and-sweep collectors; especially when the total available memory is much larger than the amount of active objects [1]. These collectors avoid touching the garbage and their work is proportional to the number of accessible pages; they do, however, split the memory into two semi-spaces halving the memory available for objects. However, objects in *EZ*'s virtual address space do not move; this restriction greatly simplifies the implementation of the persistent address space, but disallows the use of copying collectors.

THE COLLECTOR FOR DISTRIBUTED EZ

The distributed mark-and-sweep garbage collector for distributed *EZ* runs continuously without interfering with the rest of the system, and performs one collection after another. Identical copies of the collector run on each processor in the system; technically, there is one collector for each memory manager. Since the address space is distributed among all processors, each collector can access only some virtual-memory pages. In particular, each collector processes the pages owned by its associated manager, which avoids page migration — or even remote reads — for garbage-collection purposes only.

Like uniprocessor mark-and-sweep collectors, *EZ*'s collector marks all pages that hold accessible objects starting from a few system root objects. The algorithm also marks pages referenced from within marked pages, which may cause some inaccessible pages to be marked. Indeed, the collector is conservative: it marks a superset of the accessible pages and collects only a subset of the inaccessible ones [7]. It repeats the collection continuously, so it eventually reclaims all inaccessible pages.

Most pages hold only one *EZ* value or internal data structure; conversely, all aggregate values use an integral number of pages. For example, strings are implemented by lists of pages and arrays are implemented as pointers to pages of additional pointers to pages of values. These techniques produce a system in which pages cannot have both accessible and inaccessible objects; when a value becomes inaccessible, all of its pages become inaccessible. ¿From the point of view of memory allocation, these techniques eliminate external fragmentation and simplify memory management, which deals only with single-sized pages; however, small aggregate objects produce internal fragmentation because they do not use all the memory allocated to them; a small page size reduces this effect. ¿From the point of view of garbage collection, the techniques allow collection in a per-page basis, with the certainty that no page can be marked solely because of a gratuitous reference from an inaccessible object that happens to reside in an accessible page.

The mark phase of each local collector proceeds in cycles. After marking the accessible owned pages, the local collector broadcasts the set of interprocessor references found in those pages to all other collectors; the collector for distributed Smalltalk [17] performs a similar exchange. When it receives this same information from the other collectors, the local collector learns about remote references to its local pages. This information feeds another marking cycle that expands the local collector's set of accessible pages. The exchange of information keeps the collector cycles synchronized, but the mutators need not halt. The mark phase ends when, after an exchange of information, no collector expands its accessible-page set.

During the sweep phase, the collector must check whether or not each local page is marked. The implementation, however, does not require access to the pages themselves. The collector accesses only its own local data structures, which fit in a relatively small number of pages, to determine which local pages are unmarked. The sweep phase requires no interprocessor communication.

MUTATOR COLLABORATION

A concurrent garbage collector requires some form of mutator cooperation to mark the reachable objects properly; the goal of "no extra overhead for the mutator" is unattainable [7]. Without adequate protection, interleaving the execution of the collector with the mutators could produce some undesirable effects. In particular, when a reference is updated, either the old target or the new target must be marked atomically. Failure to do so may cause the collector to reclaim either the old or new target erroneously. Most concurrent collectors mark the new target to avoid hanging on to the page holding the old target [7].

EZ's collector, however, marks the old target for two reasons. First, marking the new target requires direct mutator assistance. Second, marking the old target permits the collector to use the virtual-memory system to mark the pages referenced by a page before it is modified; alternatively, virtual-memory hardware could be used. At the start of a collection, all owned pages are set to read-only. The first write to a page causes a page fault, and pages referenced from the faulted page are marked before the fault handler approves write access to the page. Marking the referents of a page *before* it is updated is equivalent to marking the old targets of updates.

With this scheme, no direct mutator assistance is required by the collector. The virtual-memory manager accomplishes the desired effect; the mutator processes themselves need not be modified in any way. However, this approach implies that garbage produced during one collection will be collected only in the next collection, as in the distal-objects collector [16].

THE COLLECTION ALGORITHM

Figure 1 shows pseudo-code for *EZ*'s garbage-collection algorithm. At every moment, each memory page is marked with one of three possible colors: white, grey, or black. Stop-and-collect uniprocessor collectors that are not concurrent use only two colors; the third, grey, labels pages that have been marked, but not yet traversed for pointers. At the beginning of each collection, locally owned pages have the same color, say, white. After the mark phase, pages are colored either white or black, and white pages can be reclaimed. Initially, `retain` and `gather` are black and white, respectively. The colors reverse roles in subsequent collections.

Each collection starts by setting owned pages to be read-only, so that their referents are marked before the pages are modified. The memory manager `scan`s the requested page before granting the access. This action is the only interaction between the mutators and the collector. As shown in Figure 1, scanning a page `shade`s its referents, colors the page black, and unprotects it.

Then, the referents of the local roots are colored by `Shade`, which colors owned white pages grey and adds remote pages to its argument `s`. When shading the referents of the local roots, the argument is the set `remote`, which is empty initially and is used by the algorithm to record all remote pages that are locally referenced and should, therefore, be marked.

Next, the mark phase begins to cycle. In each cycle, locally owned grey pages are scanned. Scanning a page may yield another grey page, but this activity ends eventually. Once all grey owned pages are scanned, all pages reachable from the local roots have been colored black, and `remote` contains all remote pages that should have been colored grey in this cycle.

A subset of `remote` is then broadcast to the other processors. This subset is the set of pages that have not been announced in previous broadcasts. Variable `cycle` records the size of this subset, and `remote` is added to `all`, which accumulates all accessible pages in the address space known to this collector. The collector then consumes similar messages from the other collectors, accumulates their sizes in `cycle`, and shades the pages mentioned in the messages. When shading the pages mentioned by other collectors, `Shade` colors owned white pages grey and adds remote pages to the set `all` directly.

Messages from other collectors reveal information about local pages that are referenced remotely and feeds another marking cycle, just as if the root set had been augmented. The new cycle might expand the local collector's set of accessible pages or its set of interprocessor references. Since each collector waits for the messages from all other collectors to arrive, these messages not only communicate the remote references between collectors, but also synchronize them and ensure that they cycle in lock-step. All collectors see the same messages in each cycle.

As the mark phase progresses, each collector's `all` becomes larger until `remote-all` becomes empty, i.e., until all grey pages everywhere have been colored black. All collectors then send an empty message, and all realize simultaneously — i.e., at the end of the same cycle — that the mark phase has finished. At that point, all owned white pages are added to the set of free pages, the roles of white and black are reversed and collection begins anew.

```
retain, gather $\leftarrow$ BLACK, WHITE
all $\leftarrow$ remote $\leftarrow$ $\emptyset$
do forever
    for every p $\in$ owned do
        Access(p) $\leftarrow$ READ_ONLY
    for every reference r in the local roots do
        Shade(Page(r), remote)

    do
        while $\exists$p $\in$ owned $\wedge$ Color(p) = GREY do
            Scan(p, remote)
        cycle $\leftarrow$ |remote - all|
        broadcast remote - all
        all $\leftarrow$ all $\cup$ remote
        for every other processor P do
            receive message k from P
            cycle $\leftarrow$ cycle + |k|
            for every p $\in$ k do Shade(p, all)
        remote $\leftarrow$ $\emptyset$
    while cycle > 0

    for every p $\in$ owned do
        if Color(p) = gather then
            free_pages $\leftarrow$ free_pages $\cup$ {p}

    all $\leftarrow$ $\emptyset$
    gather, retain $\leftarrow$ retain, gather


Scan(p, s):
    for every reference r $\in$ p do
        Shade(Page(r), s)
    Color(p) $\leftarrow$ retain
    Access(p) $\leftarrow$ READ_WRITE


Shade(p, s):
    if p $\in$ owned then
        if Color(p) = gather then Color(p) $\leftarrow$ GREY
    else
        s $\leftarrow$ s $\cup$ {p}
```

Figure 1: Garbage-collection algorithm.

When a page migrates, it is scanned *before* being sent to its new owner, if necessary. The memory manager triggers the scan by simulating a local write, which causes the page to be scanned. The new owner colors the page black. The memory manager also collaborates to the garbage-collection process when allocating free pages; they are colored black before being allocated.

IMPLEMENTATION

The implementation of the collector for distributed *EZ* has a single thread per processor for the garbage-collection proper, but every message received from the other collectors is processed by an independent thread. In addition, threads from the virtual-memory manager and interpreter threads call collector routines.

The collector thread pauses periodically to reduce the time it interrupts the mutators. The mark phase pauses after shading each of the local roots, and after each grey page is scanned. When sending the messages to the other collectors at the end of each cycle, the collector pauses after broadcasting each set of references that fits in a page. The sweep phase pauses after determining the free local pages in each page of the bit map for the set of owned pages.

The only remote communication induced by garbage collection is the message exchange with the other collectors that takes place at the end of each cycle in the mark phase. There is no migration of pages or remote accesses due to garbage collection. The collector thread does get read access to local pages in order to scan them.

The collector uses reasonably efficient representations for all its data structures, which are kept in the persistent address space. The set of available pages, `free_pages`, is represented as a list of page ranges stored in pages. Similarly, page sets for `k` and `remote` are implemented as lists of pages containing page numbers; management of these data structures is simple because they are merely bags of page numbers in which pages may appear more than once.

The implementation of marks differs slightly from the pseudo-code shown in Figure 1. Marking pages with three different colors is wasteful because it requires two bits per page, although two bits can distinguish four different states. Instead of being colored, pages are simply marked or not. A private bit map is used by each collector to keep the marks for all pages in the address space. A bit is set in the bit map if the respective page is marked, i.e., it has been processed; the bit is clear if the page has not been scanned. A marked page is equivalent to a page that has been colored black. An unmarked page, which has its bit clear, can be either white or grey.

In the algorithm, grey pages are those that have been recognized as accessible, but not yet scanned. The implementation keeps a bag with the local grey pages, in the same way it keeps the bag, `remote`, of remote pages that are locally referenced. When a page from this bag is scanned, the collector also sets its mark bit. Since only unmarked pages are scanned, no page is scanned more than once, even if they appear again in the bag.

The bit map for marked pages is also used to mark remote pages that, through the messages received from other collectors, are known to be accessible. With this extension, this bit map also represents the set `all` in the pseudo-code, which need not be implemented separately.

The implementation makes `Shade` simpler and faster because it does not have to determine the color of owned pages. When the collector shades a page, it inserts the page's number in one of two bags depending on whether or not the page is local. It inserts local pages into the bag of grey pages and remote pages into the bag `remote`. Remote pages can also be marked immediately. Pages listed in a message received from another collector, are treated similarly. Local pages are inserted into the bag of grey pages and remote pages are simply marked.

The set of owned pages, the set of available pages, and the per-page data are shared by the collector and the memory manager with appropriate locking. The data structures private to the collector — the bit map with marks, the bag of grey pages, and the bag of remote pages that are locally referenced — are not re-used, but allocated each time they are needed. The garbage collector simply loses their addresses at the end of a collection, and they become garbage to be collected by a subsequent collection. This mechanism has several benefits. No re-initialization phase is necessary before being re-used, which would create synchronization hazards with the mutator threads and the threads processing messages from the other collectors. More importantly, the collector need not process these structures because their pages are marked when allocated and cannot be reclaimed by the current collection; thus, garbage-collection time is reduced. Finally, it eliminates the implementation problems that appear when the structures are processed by the collector itself due to mutual recursion among different parts of the system.

PERFORMANCE

A prototype has been implemented on a SPARC processor — a Sun 4 work-station — running the Sun Operating System version 1, release 4.1.1-GF. The prototype runs on a single processor and uses UNIX processes to simulate different processors.

Four sequential and one parallel programs were used to evaluate the system. To simulate the effects of sharing, the sequential programs were compiled on one processor and executed on a different one. Pages would migrate from the former to the latter processor as needed for execution of the programs. The parallel program was compiled on one processor and its threads were run on each available processor. These threads actively accessed some common data structures to read and write values. The pages that contain these structures were required for read and write access from different processors.

The measurements obtained with the prototype show good performance of the distributed mark-and-sweep garbage collector [5]. It integrates well with the memory manager and rest of the system. Collections are reasonably fast, and many could be performed while running the test programs. The collector is especially effective when the system activity is really distributed among several processors; it takes advantage of performing mostly local operations and runs without increasing the running time of the mutators significantly.

The number of cycles of a mark phase is important because it is a measure of the interprocess-communication activity induced by the garbage-collection process. The actual data structures used by the programs, and its interrelations, help determine the number of cycles. The measurements show that very few cycles are required for the test programs and that, consequently, the communication overhead due to the collector is small. On average, the mark phase of the collector needs fewer than 3 cycles per collection, and no collection requires more than 5 cycles.

Most pages scanned by the collection algorithm are scanned by the collector thread; only a few pages are scanned for the mutators by the memory manager. Consequently, few page requests are delayed by the scanning process. In most cases, the time required to gain access to a page is due to the costs of memory management itself.

References

[1] Andrew W. Appel. 1991. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation Techniques*, chapter 4. MIT Press.

[2] Andrew W. Appel, John R. Ellis, and Kai Li. 1988 (July). Real-time concurrent collection on stock multiprocessors. *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7):11–20.

[3] L. Augusteijn. 1987. Garbage collection in a distributed environment. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages (LNCS 259)*, pages 75–93, Berlin. Springer-Verlag.

[4] Anders Bjornerstedt. 1989. Secondary storage garbage collection for decentralized object-based systems. In D. C. Tsichritzis, editor, *Object Oriented Development*, pages 277–319, Geneve: Centre Universitaire d'Informatique.

[5] Alvaro E. Campos. 1993 (June). *Distributed, Garbage-Collected, Persistent, Virtual Address Spaces.* PhD thesis, Princeton University, Princeton, NJ.

[6] Alvaro E. Campos and David R. Hanson. 1992 (September). Distributed *EZ*. In *Proceedings of the 16th Annual International Computer Software and Applications Conference*, pages 136–142, Chicago, IL.

[7] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978 (November). On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975.

[8] Christopher W. Fraser and David R. Hanson. 1985 (January). High-level language facilities for low-level services. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 217–224, New Orleans, LA.

[9] Robert H. Halstead, Jr. 1984 (August). Implementation of Multilisp: Lisp on a multiprocessor. In *ACM Conference on LISP and Functional Programming*, pages 9–17, Austin, TX.

[10] David R. Hanson and Makoto Kobayashi. 1990 (March). *EZ* processes. In *Proceedings of the International Conference on Computer Languages*, pages 90–97, New Orleans, LA.

[11] Suresh Jagannathan and Jim Philbin. 1992 (June). A foundation for an efficient multi-threaded scheme system. In *ACM Conference on LISP and Functional Programming*, pages 345–357, San Francisco, CA.

[12] Bernard Lang, Christian Queinnec, and José Piquer. 1992 (January). Garbage collecting the world. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 39–50, Albuquerque, NM.

[13] Claus-Werner Lermen and Deiter Maurer. 1986 (August). A protocol for distributed reference counting. In *ACM Conference on LISP and Functional Programming*, pages 343–350, Cambridge, MA.

[14] Kai Li and Paul Hudak. 1989 (November). Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359.

[15] Joseph Pallas and David Ungar. 1988 (July). Multiprocessor Smalltalk: A case study of a multiprocessor-based programming environment. *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7):11–20.

[16] Martin Rudalics. 1986 (August). Distributed copying garbage collection. In *ACM Conference on LISP and Functional Programming*, pages 364–372, Cambridge, MA.

[17] Marcel Schelvis and Eddy Bledoeg. 1988 (August). The implementation of a distributed Smalltalk. In Gjessing, Stein and Kristen Nygaard, editors, *ECOOP'88: European Conferece on Object-Oriented Programming (LNCS 322)*, pages 212–232, Berlin. Springer-Verlag.

[18] Niklaus Wirth and Jurg Gutknecht. 1989 (September). The Oberon system. *Software—Practice & Experience*, 19(9):657–693.