A Fast Parallel Conservative Garbage Collector for Concurrent Object-Oriented Systems*

— Extended Abstract —

Satoshi Matsuoka, Shin'ichi Furuso, and Akinori Yonezawa

Department of Information Science
The University of Tokyo
{matsu,furuso,yonezawa}@is.s.u-tokyo.ac.jp

Abstract

In future OO-OS, multiple applications written in heterogeneous languages will interact via shared objects. There, *conservative GC (garbage collection)* could provide storage management independent of programming languages and applications. Conservative GC, however, is difficult to extend naively into a concurrent scheme that meet the demands of very fast allocation — up to one million objects per second. Our *parallel conservative GC* resolves this by allowing allocation and collection to be done almost totally in parallel. We currently estimate that allocation takes less than 20 instructions, including the overhead of (rare) synchronization. The preliminary version of the algorithm is running on a 4-processor LUNA-88K with Mach 2.5. Our algorithm relies on the virtual memory primitives to be available to the user in order to maintain the necessary invariants; we briefly discuss the requirements of the future OO-OS in this respect.

1 Introduction

The most important abstraction in the future operating systems will be that of *objects*; such operating systems, as well as its client programs, will be modeled as groups of coordinating concurrent objects. Such a picture is clean and straightforward if the operating system environment supports only a single programming language, as is the case for Lisp or Smalltalk machines. However, a general purpose operating system usually needs to be language independent, for it must support the heterogeneous execution models of multiple languages. There, one must provide appropriate abstractions that serve as a common platform for multitudes of different languages. *Lightweight thread* is one popular abstraction; recently, more elaborate abstractions of programming languages are being introduced, such as the recent work in Mach that allow the user to treat *continuations* in the style of Scheme as language independent first class data[5]. For objects, the usual common notion is that they serve to encapsulate some abstraction; from the implementation standpoint, they physically occupy some storage in virtual memory, and are accessed via some logical object IDs, or direct physical pointers when multiple applications share a common address space.

Now, many modern-day programming languages require automatic storage management of objects, particularly their automatic reclamation, usually provided with some sort of built-in *garbage collection* (hereafter GC). When concurrent objects are used as common abstraction among multiple distinct languages, however, the collectors may not work correctly if the objects are shared among them. This is because each language will have its own storage management mechanism built-in. In a sense, the abstraction of storage management provided by the current operating systems is not enough to allow multiple heterogeneous languages to interact via the concurrent objects abstraction. In fact, even for the case of a single language, the GC algorithm might behave incorrectly when two separately compiled programs are loaded into a single address space for coordinated execution.

Conservative GC, originally proposed by Boehm and Weiser[4], was a major step towards providing an abstraction for automatic object storage management *independent of programming languages and ap-*

*In Proceedings of IEEE International Workshop on Object Orientation in Operating Systems, Palo Alto, CA, Oct. 1991, Luis-Felipe Cabera, Vincent Russo, Marc Shapiro eds., IEEE Computer Society Press, pp. 87–93.

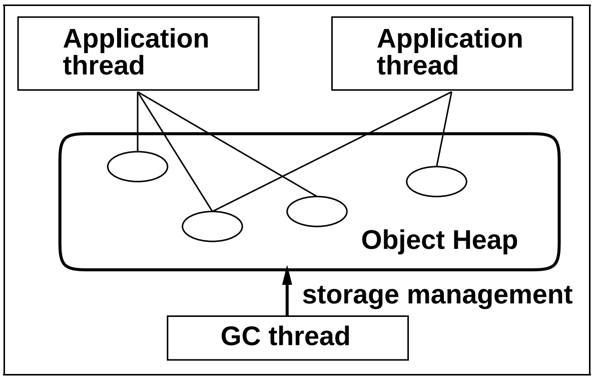

Figure 1: Garbage Collector Threads and Application Threads

plications. The problem, however, is that the Boehm-Weiser algorithm is a stop-and-collect algorithm, and is difficult to extend naively into a concurrent scheme, where allocation speed of up to one million objects per second per thread is required. Such difficulty is understandable, because by nature conservative GC does not assume any help from languages or applications (called *mutators* in the GC terminology), whereas detailed run-time information of mutator action is required for all concurrent GC algorithms to work correctly. The brute-force approach would then be to forego concurrency by stopping all the executing threads, and merely perform the stop-and-collect conservative collection for the frozen state — such an approach, however, is highly undesirable in the current day operating systems, whose concurrency and real-time properties cannot be sacrificed.

2 Our algorithms for fast Parallel Conservative GC

Based on the above observations, we have devised two garbage collection algorithms, *basic* and *extended*, that meet the requirements of being (1) language independent (conservative), as well as being (2) parallel (and real-time), and (3) implementable on stock architectures. Both of our algorithms can be executed in user-space of operating systems running on a shared-memory architecture.

In both the basic and extended algorithms, there are application threads and GC thread(s), who have common shared accesses to the object heap (Figure 1).

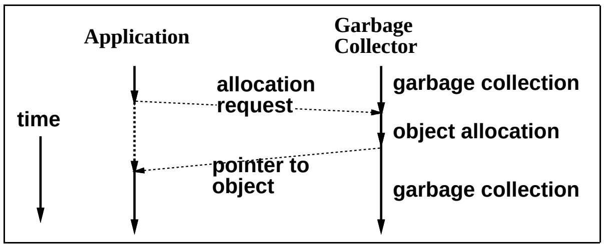

Figure 2: Interaction between GC and application threads

2.1 The basic algorithm

In the basic algorithm, each application thread interacts with the GC thread for object allocation. In the current implementation on Mach, we employ the inter-thread message sending of the C-threads library. The GC thread runs totally in parallel with the application threads on a shared-memory machine. Figure 2 shows the timing chart of the interaction between GC and application threads: the application thread requiring memory allocation sends an allocation request message to its associated GC thread. The GC thread, upon receiving the message, allocates the object in the heap and returns its reference to the application. The application, in turn, resumes execution; the GC thread continues with garbage collection activities.

The garbage collector runs incrementally with an algorithm similar to the Yuasa real-time garbage collector[10]. To avoid the contention between GC and application threads, we take a *virtual snapshot* (using the copy-on-write mechanism in Mach) of the object heap just before entering mark phase, in a similar manner to the interleaving-concurrent conservative algorithm briefly mentioned in [6]. In essence, the GC threads run in the past image of the virtual snapshot in the mark phase, whereas the mutator is running on the present image. In the sweep phase, the objects are collected in the current image based on the marking information of the past image. During the entire mark-sweep cycle, the total stoppage of all the application threads plus the synchronization among the GC threads is only necessary when the virtual memory tables are initially prepared for copy-on-write. During the rest of the cycle, the threads synchronize with the GC thread(s) only on individual basis.

The correctness of the basic algorithm could be understood intuitively, although the detailed proof is unfortunately tedious and cumbersome[1]. In particular, the following lemma (described only informally here)

[1]Subject of our forthcoming paper.

plays the chief role role:

Lemma 1 (informally) *Any object that is a garbage in the past image is a garbage in the present image, unless it is re-allocated by the allocator.*

The proofs of the above lemma and others are important, because they describe some invariant properties that are essential for the basic algorithm to evolve into the extended algorithm, as we we will present in the next section.

We have already implemented the basic algorithm on several machines, such as LUNA 88K (MC88100 CPU $\times$ 4, Mach 2.5, C Threads) and NeXT (MC68040 CPU, NeXT OS 2.0 (Mach 2.5), C Threads). Almost all concurrent programs under C Threads can be garbage collected with only trivial modifications. A concurrent OO-language in particular is well-suited for this collector, because it is usually well-behaved in the sense that there is little chance of violating the necessary conditions for the conservative GC to work correctly, compared to raw C programs which have far greater liberty with pointer operations.

2.2 Problems with the basic algorithm

The benchmarks with the basic algorithm, however, revealed the following problem: When multiple application threads contend for object allocation, the overhead of synchronization and control transfers among the threads overwhelm the execution of application and GC threads even for allocation speeds in the order of 10^3 object allocations per second per thread in the worst case. Consequently, the garbage collector becomes a total bottleneck for concurrent languages with frequent memory allocations, such as MultiLisp[7] and ABCL/1[9], which could allocate as much as one million objects per second per each thread.

The benchmarks with the basic algorithm, however, revealed the following problem: the overhead of object allocation is very high when multiple application threads contend for object allocation. Our benchmarks show that in an extreme (and contrived) case, the allocation speed degrades down to the order of several KOAPS (Kilo Object Allocations Per Second). In usual cases, there is a factor of 10 to 100 improvement, but that is still insufficient for highly parallel concurrent OO-languages such as MultiLisp[7] and ABCL/1[9], where allocation speed of up to one MOAPS (Million Object Allocations Per Second) could be required. Consequently, the garbage collector becomes a bottleneck for concurrent languages with frequent memory allocations; this is due to the following reasons:

1. As Figure 2 shows, GC threads are responsible for both garbage collection and object allocation. As a result, the request of an application thread for a new object is blocked if the request for another thread is being processed.
2. Sending a message between threads itself is very costly, especially on multiprocessors, since it requires costly inter-processor synchronization and control transfer with semaphores or busy waiting.

For these reasons, the system performance would degrade to the level where the performance increase achieved with parallel execution could become worthless. Thus, we concluded: although the basic algorithm is effective for systems that are pseudo-parallel and/or whose memory allocations are not excessively frequent, we have found it to be not too effective for highly concurrent systems with continues object allocators.

2.3 Extended algorithm — the primitive idea

To overcome the deficiencies of the basic algorithm, we have devised an algorithm which we refer to as the *extended algorithm*. In the extended algorithm, synchronization and control transfers among the threads are now almost totally eliminated, and as a result, the application threads and the GC threads runs continuesly in parallel with no interactions in need for synchronization and mutual exclusion.

The basic idea is to assign a individual allocator to each thread (or a group of threads). Individual allocator is then given a chunk of memory to perform the allocation, from where the objects are allocated in small pieces. Thus allocators and the collectors require less synchronization, eliminating the costly busy-waiting or semaphore waiting. This technique is similar to the one employed in Halstead's extension of Baker's collector[7], and Appel's generational collector on stock processors[1]. In our case, however, there are various differences and difficulties, such as:

1. The chunks will be exclusive to an allocator with respect to allocation only; other applications threads, as well as the collector threads, will be freely reading from and writing into the chunk in parallel to allocation.
2. The management of assignment of chunks without causing bottlenecks is a delicate issue: When the chunk is used up, a new chunk must be assigned to the allocator. If the old chunk is freed

from assignment, it will be subject to GC, after which it could be re-assigned to some other allocator if it contains enough free objects. This requires that the mark bits of the objects of the chunk to be properly managed, which in turn would cause an access conflict between the allocator and the collector. Explicit locking of the mark bits to solve the exclusion problem is naturally undesirable, for it would require lock-unlock operations per each allocation.

A simplistic scheme to resolve the above issues for parallel conservative collectors would be to assign the chunk on request from the allocator, and maintain that assignment until the applications themselves terminate. A collector can then be exclusive to each allocator, and interleave between application/allocation versus collection by making the switch at a place with respect to both 1 and 2 above (such as at the time of application request for allocation). Such a scheme, however, is not desirable for the following reason: Conservative collectors require the objects with different sizes to be allocated to different chunks, thereby requiring the allocator to hold a large set of chunks if the thread allocates objects of many different sizes, especially for the sizes that are often used. But a group of coordinating applications may contain hundreds of threads or more; thus the memory usage, even in the presence of GC and virtual memory, would become prohibitive. Also, (1) even with this scheme the allocation overhead is still substantial, and (2) the real-time characteristics of the collection algorithm is lost.

Overall, mangagement of chunks must be carefully done in order not to re-introduce the need for synchronization and/or mutual exclustion, (which we tried to avoid in the first place,) so that mutator threads and collector threads will continue to run in parallel. In the next section, we describe how this is possible.

2.4 The details of the extended algorithm

As indicated earlier, we let each application thread have its own object allocator, to which (page-sized) memory chunks are exclusively assigned[2]. To reduce the number of assigned pages, at most one page-sized chunk is assigned for each (normalized) size of object the application thread uses. Further reduction of the number of assigned pages is possible by employing various techniques (these will be described in the full version of this paper.)

[2] For objects exceeding the size of the page, a chunk consists of multiple pages

Figure 3 illustrates the structure of the chunks in our algorithm. Each chunk holds objects of only one size, and has its own internal free-list. Chunks are divided into four categories: *assigned chunks*, *waiting chunks*, *used-up chunks*, and *free chunks*. Except for the assigned chunks, they are kept in the pools (or lists) of respective types. (For waiting chunks and used-up chunks, the pool is actually further subdivided according to its internal object size for efficiency.)

Each allocator accesses the internal free-list of the assigned chunks for allocation. Waiting chunks contain at least one allocated object and at least one object in its internal free-list. Upon request from the application, the waiting chunks are assigned to the allocators. Used-up chunks are those whose internal free-list is empty. When the allocator consumes the entire free-list of the assigned chunk, it is added to the used-up chunk pool. The free chunks contain no allocated objects.

The behavior of the GC thread(s) is (are) as follows:

- **The Mark Phase:**
  1. It merges the 'used-up chunk pool' to the 'waiting chunk pool'.
  2. It suspends all the application threads and takes a *virtual snapshot* of the object heap.
  3. It then resumes the application threads and begins to trace and mark active objects in the snapshot from the execution contexts of the application threads (e.g., registers and contents of threads stack). The trace/mark is done in parallel to the application threads in a manner similar to the basic algorithm. The difference is that the garbage collector marks the live objects only in the waiting chunks. Live objects in used-up chunks and assigned chunks are traced but not marked.
- **The Sweep Phase:**
  1. The GC thread sweeps and reclaims the garbage objects in 'waiting chunks' fully in parallel to the application threads. Note that, during both phases, the application threads need to be assigned chunks that have been already swept. This is because the GC thread does not sweep assigned chunks. In the event that chunks are not available from either the waiting chunk pool or the empty chunk pool, the allocator itself sweeps one of the used-up chunks for its own use. Thus

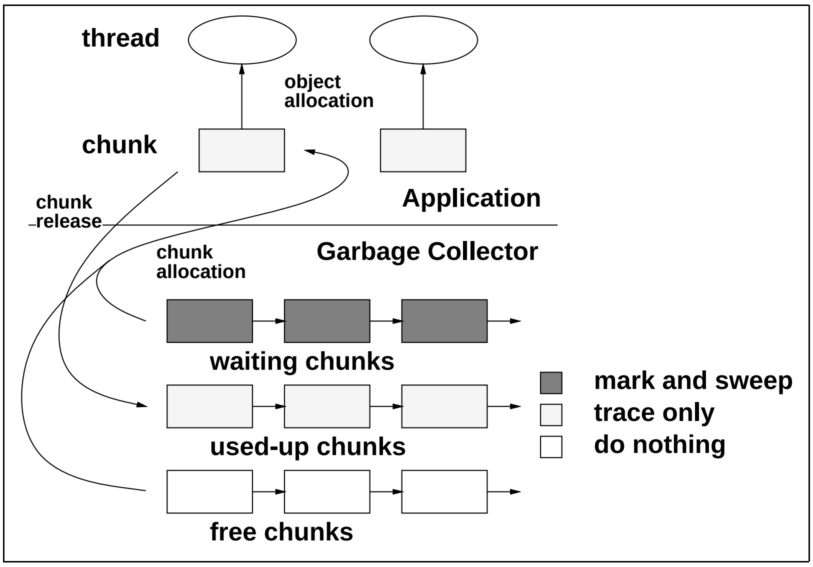

Figure 3: Chunks for the Extended Algorithm

Behavior of the allocators are as follows, in both the mark and sweep phases:

1. The allocators allocates objects from its assigned chunk. It need not interact with the GC thread for object allocation. Furthermore, it DOES NOT operate on the mark bits of objects, nor perform tracing from the newly allocated objects (neither by itself or by the GC threads). In fact, when tracing reaches a newly allocated object, it is stopped there and no further tracing is performed for the objects pointed to by the (newly allocated) object[3]. The safety here is guaranteed by the property induced by the lemma described informally below.

2. When the internal free-list of the chunk is empty, the allocator stores the chunk in the used-up chunk pool, and assigns itself a new chunk from either the waiting chunk pool or the free chunk pool. As indicated earlier, the allocator itself does sweeping in the event that no suitable chunks are available.

The distribution of allocators effectively eliminates almost all of the necessary synchronization and mutual exclusion from the algorithm. However, it raises various touchy issues with respect to its correctness. For example, is it safe for the individual allocators to allocate the objects unknowingly to the GC thread? (Note that, In the basic algorithm, the GC thread had total knowledge of the allocations it made.)

Since the garbage collection algorithm takes a virtual snapshot of the object heap in the same manner as the basic algorithm, the following lemma proven:

Lemma 2 (informally) *Objects newly allocated during the mark phase need not be traced, if they are themselves not subject to collection.*

The lemma follows because we can show by induction that, for every newly allocated objects, the objects that are elements of its transitive closure of references are either: (1) old objects that will be properly marked during the mark phase by some other reference, and (2) (other) objects newly allocated during the mark phase. Thus, if we do not have the newly allocated objects subject for collection, the allocator need not operate the mark bits nor notify the garbage collector about the allocation. The only locking required is when chunks are assigned from/stored into the chunk pool.

We are now implementing the extended algorithm on LUNA 88K using C-Threads, as well as formulating a formal proof of its correctness based on the proof of the basic algorithm. In comparison to the basic algorithm, the extend algorithm will be substantially efficient with respect to object allocation. Since there are no synchronization required for the usual case (when an object is allocated from the internal free-list), we expect our extended collector to be able to allocate

[3] As to *how* we determine that a given object is a newly allocated one is a delicate issue; we skip the details here and merely state that this can also be done with no mutual exclusion involved

objects in approximately 10 – 20 instructions in those cases. Furthermore, due to the shortness of the allocation code, we could even allow its *inlining* for further improvement in performance.

To estimate the efficiency of our extended algorithm, assume that a language implementation such as Lisp allocates objects every 50 instructions, and the allocation takes 10 instructions. Assume also that the object size is 8 bytes (the size of the Lisp `cons` cell), the page size is 4096 bytes, and a page is 75% empty when assigned to an allocator. Then, the number of instructions executed by the processor until the page is filled is:

$$4096/8 * .75 * (50 + 10) \simeq 23000 \text{ instructions.}$$

Supposing that the the allocator action to obtain the fresh page is approximately 1000 instructions on the average, the overhead of allocation for our extended collection, compared to allocation in sequential systems, is:

$$1000/(23000 + 1000) \simeq 4\%,$$

or, approximately 3 instructions per allocation. Thus, we could say that the expected overhead in allocation due to concurrency is minimum. For the collector threads, we do not have estimates on the amount of computation required; however, based on the experience with our basic algorithm, we expect that the cost to be well within a reasonable range.

3 Parallel Conservative GC and concurrent OO-systems

We briefly outline the requirements on the operating systems to realize our algorithm, as well as the features desirable to be available to gain even more performance improvements and achieve better integration of various algorithms dependent on the availability of virtual memory primitives.

In both algorithms, due to the necessity of detecting write faults, the appropriate virtual memory primitives must be made available to users by the operating system. The problem with many current-day operating systems, however, is that virtual memory primitives are not efficiently supported; in fact, there is a three- to four-fold difference between the fastest and the slowest in performance for the operating systems on the same architecture[2]. It is important for the operating systems of the OO-concurrent systems to handle user-level virtual memory operations efficiently for efficient execution of distributed algorithms, such as GC, distributed shared memory, and checkpointing[2].

Another problem is that combination of multiple algorithms dependent on them are currently very difficult to be done in an orthogonal, portable way in current operating systems. For example, there is no easy way to orthogonally combine our GC algorithm with, say, a simple heap range-check algorithm. We contend that the operating system should have proper mechanisms so that multiple handlers of page fault traps are appropriately coordinated.

Further efficiency could be attained in collection if either of our algorithms is made part of the object storage and virtual memory management facility of operating systems. Another possibility is to integrate the GC facility with the persistent object system of the secondary store.

4 Related work

Appel's copying collector[1] was perhaps the first practical concurrent collector on a stock architecture. They too, overcome the efficiency issues of maintaining the invariants in concurrent collection using virtual memory primitives. The difference is that Appel's collector is an extension of the Baker's copying collector; therefore, the use of virtual memory (and the algorithm itself) is totally different from ours. In particular, they employ virtual memory primitives to detect references into the *from-space* by the mutator, whereas we employ the primitives to take a virtual snapshot of heap storage. As a result, only the write fault detection is needed, once per page per each mark-collect phase in our algorithm.

The interleaving concurrent conservative algorithm is mentioned in [6] very briefly. The basic idea seems very similar to our basic algorithm in that it also employs the virtual snapshot technique. Mutation and tracing/collection are done in an interleaved fashion, however; several steps of incremental tracing is performed per each allocation request by the mutator, after which the control is returned to the mutator. Therefore, the mutator has exclusive access to the heap during the mark phase. (NOTE: Although it is difficult to judge precisely due to the abovementioned briefness, our *basic* algorithm might be a direct parallelization of their incremental algorithm — in such a case, our contribution for the basic algorithm would be as follows: (1) direct parallelization from the incremental version requires a proof that there are no critical regions between the collector and the mutator threads: although we restricted ourselves to an informal overview of the proof, we plan to present its full formulation in the full version of this paper; and (2) we have actually implemented and measured the performance of the algorithm, thereby identifying its weakness which led to the development of our extended algorithm. The extended algorithm itself is of course totally different.)

The latest work in [3] describes a mostly parallel collector which can naturally be extended into a conservative algorithm. It too employs the virtual memory primitives of the underlying OS. The advantage of the algorithm is that it does not involve the copying of the page upon write fault. The disadvantage is that it is not a fully parallel algorithm, requiring a *finalization stage* where all the threads must be stopped for exclusive access by the collector. For highly parallel applications, the finalization stage could take substantial amount of time. Although the intermediate tracing before the finalization could reduce the the duration of finalization to a notable degree for pseudo-parallel systems, it remains to be seen whether a similar reduction could be achieved for highly parallel systems.

5 Future work

Currently, our algorithms run on shared memory architectures; we believe the extended algorithm is well suited for further extensions to Distributed Shared Memory, due to the several favorable characteristics:

1. because the allocator is distributed and concentrates on allocating from a single page for each object size, the processor won't thrash on the newly allocated objects,

2. the collector threads can be distributed naturally to processors so that they manage the physical pages of each processor, and

3. the handling of mutator writes into the page can be handled simultaneously with the consistency maintenance protocol for shared pages, particularly the many-reader, single-writer consistency algorithm[8].

We also expect that our extended algorithm can be merged with the generational extension to conservative GC by Demers et. al.[6]; we would then have almost an ideal GC for concurrent object-oriented systems, being conservative, fully parallel, and very efficient with respect to both allocation and collection.

References

[1] Andrew W. Appel, John R. Ellis, and Kai Li. Real-Time Concurrent Collection on Stock Multiprocessors. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20. ACM, 1988.

[2] Andrew W. Appel and Kai Li. Virtual Memory Primtives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Langauges and Operating Sytems (ASPLOS)*, pages 96–107. ACM, 1991.

[3] Hans Boehm, Alan Demers, and Scott Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164. ACM, 1991.

[4] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooporative Environment. *Software—Practice and Experience*, 18(9):807–820, ,sep 1988.

[5] Richard P. Davis, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Continuations: Unifying Thread Management and Communication in Operating Systems. Technical Report CMU-CS-91-115, School of Computer Science, Carnegie Mellon University, March 1991.

[6] Alan Demers, Mark Weiser, Barry Haynes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining Generational and Conservative Garbage Collection: Framework and Implementations. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 261–269. ACM, 1990.

[7] Robert H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Compuation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–517, 1985.

[8] Michael Strumm and Songnian Zhou. Algorithms Implementing Distributed Shared Memory. *ieee-computer*, 23(5):54–64, May 1990.

[9] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. Computer Systems Series. The MIT Press, 1990.

[10] Taiichi Yuasa. Real-Time Garbage Collection on General-Purpose Machines. *Journal of Systems and Software*, 11(3), March 1990.