Mostly Parallel Garbage Collection

Hans-J. Boehm
Alan J. Demers
Scott Shenker

Xerox PARC

Abstract

We present a method for adapting garbage collectors designed to run sequentially with the client, so that they may run concurrently with it. We rely on virtual memory hardware to provide information about pages that have been updated or ‘‘dirtied’’ during a given period of time. This method has been used to construct a mostly parallel trace-and-sweep collector that exhibits very short pause times. Performance measurements are given.

1. Introduction

Garbage collection is an important feature of many modern computing environments. There are basically two styles of garbage collection algorithms: reference-counting collectors and tracing collectors. In this paper we consider only tracing collectors. A straightforward implementation of tracing collection prevents any client action from occurring while the tracing operation is performed. When applied to a system with a large heap, such stop-the-world implementations cause long pauses. One of the primary arguments against wide adoption of garbage collection is that these collection-induced pauses are intolerable.

There are two common approaches to reducing the pause time in tracing collectors: generational collection, and parallel collection.

 Appeared in *Proceedings of the ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation SIGPLAN Notices 26*, 6, pp. 157-164.

Generational garbage collectors concentrate on reclaiming recently allocated objects. Generational collectors have been implemented in a wide variety of systems and have achieved significantly reduced pause times [Ungar 84]. However, a generational collector still needs to run full collections occasionally in order to reclaim older objects. Thus, the problem of long pauses is not completely eliminated.

Parallel collectors take an orthogonal approach to the problem of reducing collection pause time. Rather than decreasing the total amount of work performed during a particular collection as generational collectors do, parallel collectors merely mitigate the effect of this work by running in parallel with the mutator (client). While a parallel collector still imposes some overhead cost on the system, it eliminates the long pause times associated with stop-the-world collection.

This paper discusses a technique called *mostly parallel* tracing collection. In a mostly parallel collector, some small portion of the tracing algorithm must run in a stop-the-world fashion, but the majority of the work can be done in parallel.

We had two goals in writing this paper. First was to present a method for transforming a stop-the-world tracing collector into a mostly parallel collector. This method is quite general: it applies to copying as well as non-copying collectors, and to generational as well as non-generational ones. Furthermore, its implementation makes few demands on the operating system beyond the write-protect facilities that are now widely available. Our second goal was to describe a particular implementation of a garbage collector that illustrates this idea. Combining the notion of mostly parallel tracing collection with our previous work on conservative [BoehmWeiser 88] and generational [DemersEtAl 90] collection, we have built a conservative, generational, mostly parallel collector. This collector is able to provide sophisticated garbage collection services to rather primitive languages like C which provide no pointer information. Collection pauses on a SparcStation II with 15 Megabytes of accessible objects are usually not noticeable. Unlike pure generational collectors, our collector achieves this performance even for the periodic full collections needed to reclaim long-lived objects.

2. Making Tracing Collectors Mostly Parallel

Basic Idea

Every program contains a set of *root* memory objects (machine registers, statically-allocated data, etc.) that are always accessible. A tracing garbage collection starts with an *immune set* of memory objects that includes all roots, follows the pointers contained therein to other memory objects, and then continues this pointer tracing recursively until no more objects can be reached. We use the term *marked* to denote those objects visited by this tracing procedure. Any marked memory object is *reachable* from the immune set and should be saved. Unmarked, and thus unreachable, objects are garbage and should be reclaimed. Tracing collectors can differ in the immune set used (generational), whether or not objects are moved (copying), and many other implementation details.

We believe that a wide variety of tracing collectors designed to run in a stop-the-world fashion can be made to run mostly in parallel. We first discuss this in rather general terms and then return in Section 3 to give a more precise definition for the noncopying case.

Assume we are able to maintain a set of *virtual dirty bits*, which are automatically set whenever the corresponding pages of virtual memory are written to. (An acceptable implementation of this feature can be obtained by write-protecting pages and catching the resulting write faults, with no modifications to the underlying OS kernel; an implementation in the OS kernel would of course be more efficient.) For any tracing collector defined for stop-the-world operation, consider the following collection algorithm. At the beginning of the collection, clear all virtual dirty bits. Perform the traditional tracing operation in parallel with the mutator. The virtual dirty bits will be updated to reflect mutator writes. After the tracing is complete, stop the world and trace from all marked objects that lie on dirty pages. (Registers are considered dirty.) At this point, all reachable objects are marked, and garbage can safely be reclaimed.

This requires that the tracing operation not invalidate the original data structures seen by the mutator. This is normally automatically true if objects are not moved. In the case of a copying collector, a possible approach is presented in the last section.

In this algorithm, the parallel tracing phase provides an approximation to the true reachable set. The only objects unmarked by this parallel tracing process which are indeed reachable must be reachable from marked objects which have been written since being traced. The stop-the-world tracing phase traces from all such objects, so that in the end no truly reachable objects remain unmarked. The application of this idea to noncopying collectors will be formalized in the next section.

The resulting mostly parallel collector is a compromise; it is neither perfectly parallel nor precise. The severity of these drawbacks depends on the writing behavior of the mutator. The duration of the final stop-the-world phase is related to the number of pages written during the parallel tracing operation. Thus, running this collector during a period of rapid writing could lead to long system pauses. In the worst case, pause times would be comparable to a stop-the-world collection, but this has never been observed in practice.

Not all unreachable objects are reclaimed. This occurs when a pointer which has been traced through in the parallel trace phase is changed or deleted before the stop-the-world phase. However, such an object will be reclaimed by a subsequent collection. Thus, the rate at which pointers are modified will determine the lack of precision in the collection.

Related Work

Our work is motivated by our search for effective garbage collection algorithms that can operate without any special operating system or mutator cooperation. In particular, we are interested in a collection algorithm usable by programs written in C on a standard UNIX system. Thus, algorithms that require reliable pointer identification and possibly mutator cooperation, such as copying or reference counting, were not pursued to much depth. Thus we emphasize noncopying collectors. In the next section we formalize our mostly parallel technique for this case.

There are a number of related collection algorithms that rely on copying live data and thus assume reliable pointer identification. These can generally be made to tolerate some uncertain pointer identifications using the technique of [Bartlett 89]. However, this can only accommodate a small number of uncertain pointers. It usually performs acceptably only if the uncertainty is limited to pointers in registers and on the stacks. Even then it may occasionally be problematic [DeTreville 90]. In our environment, every pointer identification is uncertain, including those from the heap, and this approach is not usable.

The advantages of being able to accommodate uncertainty in pointer identification are described in [BoehmWeiser 88] and [DemersEtAl 90]. An analysis of the limitations of the technique under very adverse circumstances is given in [Wentworth 90]. [Zorn 90] demonstrates that noncopying trace-and-sweep collectors may, at times, outperform copying collectors (though the details of his trace-and-sweep collector are quite different from ours).

Parallel noncopying collectors are described by Steele [Steele 75] and Dikstra et al. [DijkstraEtAl 78], among others. [Baker 78] presents a copying collection algorithm that is explicitly interleaved with mutator operations. Unlike our work, these algorithms rely heavily on mutator cooperation. Pointer updates, and in most cases read accesses, require the mutator to update collector data structures. These algorithms are practical on conventional hardware only under unusual circumstances. Baker's algorithm requires reliable pointer identification.

Appel et al. [AppelEllisLi 88] present a parallel copying collector intended to run on conventional machines. Their scheme, like ours, takes advantage of virtual memory hardware. Unlike our approach, they require intervention when the page on which an object resides is first *accessed* (either written or read), whereas our scheme requires intervention only when the page is first written, and then only if the operating system does not allow use of hardware dirty bits. Since their algorithm also copies list structures breadth-first, and thus does not preserve locality in list structures, this may result in a flurry of such intervention at the beginning of a collection.

[DemersEtAl 90] also describes a parallel collection algorithm based on virtual checkpoints implemented with a copy-on-write strategy. The algorithm described here does not incur the copying overhead, is typically easier to implement, and requires no additional memory.

Very recently, DeTreville [DeTreville 90] described a parallel trace-and-sweep collector which, like ours, uses virtual memory hardware instead of explicit mutator cooperation. His collector requires that slightly less work be performed while the mutator is stopped but, like the [AppelEllisLi 88] collector and unlike ours, it requires that the collector be notified on initial read accesses by the mutator. Furthermore, a single page may be protected and faulted more than once. Based on our experience with pages accessed versus pages written, we believe that our strategy would usually outperform this approach, at least in our environment. Comparable performance measurements would be useful, but difficult to obtain; they report few quantitative measurements, and those are on completely different hardware, with completely different mutators.

An overview of various proposed uses of virtual memory primitives by user programs is given in [AppelLi 91].

3. Sweeping Doesn't Matter

The following discussion will center on the mark phase of the mark-sweep collector, that is on the process of tracing through and identifying reachable objects. The sweep phase does not have a significant impact on garbage collector pause times. There is no reason to sweep the entire heap while the world is stopped waiting for the collection to complete; it is easy enough to interleave the ''sweep phase'' with object allocation.

Our collector splits the heap into blocks. Each block contains only objects of a particular size. For small objects, the size of the block is a physical page. The mark phase sets a bit for each accessible object. We then queue pages for sweeping, keeping a separate queue for each small object size.

The allocator also maintains separate free lists for each (small) object size. Whenever the allocator finds an empty free list, it sweeps the first page in the queue of ''sweepable'' pages for that object size, removes it from the queue, and restores unreachable objects to the free list.

Large object blocks are swept in large increments during allocations immediately following a collection. This requires very little cpu time, and does not force the data pages to become resident in physical memory.

The net effect of this is that garbage collection times are completely dominated by the time it takes to mark accessible objects, and are thus, essentially proportional to the amount of accessible space. Object allocation times may become rather long if full pages are scanned before an available object is found, but this effect is not noticeable in practice.

For the next three sections, we will view garbage collection as the process of marking reachable objects.

4. Formal Statement

In a previous paper [DemersEtAl 90] we formalized the notion of a *partial* collection, i.e. a collection that reclaims only a subset of all unreachable objects. We will not review that material here, except to note that these partial collections are characterized by the set ***T*** of *threatened*, i.e., potentially collectible, objects. The complement of that set, the non-collectible or *immune* objects ***I***, are the objects to be traced from as discussed in Section 2. The root set is always a subset of ***I***. Full collections have ***I***=roots, whereas partial collections have additional objects in ***I***. Generational collections are a special case of partial collections where the threatened set contains only recently allocated objects. A collection is correct if it does not reclaim any objects that are reachable, by tracing pointers, from ***I***. A way of guaranteeing correctness is to reclaim only unmarked objects and ensure that the following closure condition holds:

> ***C:*** Every object in ***I*** is marked and every object pointed to by a marked object is also marked.

A stop-the-world collection consists of the following steps: (1) stop the world, (2) clear all mark bits, (3) perform the tracing operation ***TR*** defined below, and (4) restart the world.

> ***TR:*** Mark all objects in ***I*** and trace from them.

At the completion of this process, condition ***C*** holds and we can safely reclaim all unmarked objects.

To run such a collector in a mostly parallel fashion, we (1) clear all mark bits, (2) clear all virtual dirty bits, (3) perform the tracing operation ***TR***, (4) stop the world, (5) perform the finishing operation ***F*** defined below, and (6) restart the world.

> ***F:*** Trace from all marked objects on dirty pages.

Note that here the tracing operation ***TR*** is performed in parallel with the mutator. The closure condition ***C*** does not hold after step 4, since the mutator could have written new pointers into previously marked objects. However, the weaker condition ***C'*** does hold at the end of step 4.

> ***C''***: Every object in ***I*** is marked and every object pointed to by a marked object on a clean page is also marked.

Notice that this weaker closure condition, once established by the operation ***TR***, remains unchanged by the actions of the mutator. Applying the process ***F*** to any state that satisfies condition ***C''*** will produce a state that satisfies condition ***C***.

This produces a correct mostly parallel collection. However, if the mutator has dirtied many pages during the tracing operation the stop-the-world phase can be overly long. To reduce this delay, the collector process can "clean" the dirty pages in parallel through the use of the process ***M*** applied to some set of pages *P*.

> ***M***: (1) Atomically retrieve and clear the virtual dirty bits from the pages *P*, and (2) trace from the marked objects on the dirty pages of *P*.

The previous discussion focused on a general notion of partial collection. We now turn to defining a particular generational version of a partial collection, which makes use of the mark bits for object age information. This collector is related to Collector I in [DemersEtAl 90]. Consider a partial collection where the set ***I*** is chosen to be the set of currently marked objects. Then, we know that condition ***C''*** already holds and that steps 1-3 are unnecessary. We then merely need to stop the world and run the finishing step ***F*** to complete the collection. In order to reduce the length of the delay, we perform the operation ***M*** applied to the entire heap immediately before the stop-the-world phase. Thus, a mostly parallel version of a generational collector can be described as (1) perform ***M*** on the heap, (2) stop the world, (3) perform ***F***, and (4) restart the world. Once an object has been marked, it will never be reclaimed by this generational collector. Thus, we must occasionally run full (nongenerational) collections to reclaim once-marked objects.

An alternate way of cleaning dirty pages is the process ***M'***

> ***M'*** (1) Atomically retrieve and clear the dirty bits from the pages *P*, (2) for all unmarked objects pointed to by marked objects on dirty pages of *P*, mark them and dirty the pages on which they reside.

We can substitute repeated applications of ***M'*** for a single application of ***M***. Usually ***M*** is preferable but if the ratio of virtual to physical memory is extremely large, it may make sense to run ***M'*** repeatedly in order to improve locality of the tracing algorithm.

5. Implementation Choices

The preceding section gives us tools to build a variety of collectors, but it is not obvious how to combine them. We have not made a systematic comparison of the options, but we have experimented with a few of them. This section describes some of those experiences.

The first choice is when and how to run ***M*** or ***M'*** before a partial collection. We chose not to use ***M'***, since its repeated use is likely to be much more expensive than a single execution of ***M*** in our environment. Our experience is that it for allocation intensive mutators it occasionally makes sense to run ***M*** more than once before a collection, since the initial execution of ***M*** can take some time, thus giving the mutator a chance to dirty a significant number of new pages. However, the cost involved with more than two iterations appears rarely to be justified.

Further variants of ***M*** are possible. It is not essential for correctness that ***M*** mark from roots. We maintain dirty bits for some roots, in order to reduce the number of roots that must be examined by ***F***. (In our environment, a megabyte of potential roots is common.) For reasons of convenience we clear all dirty bits when ***M*** starts. Thus we must mark from roots on known dirty pages. But marking from other roots, such as thread stacks, is optional.

We found it to be advantageous to always execute ***M*** once before a partial collection, and to run a second iteration if there was a significant amount of allocation during the first. Furthermore, letting the first iteration of ***M*** mark from all roots (other than those known to be clean) can significantly reduce final pause times.

A more difficult decision is what constitutes a full collection, and how and when we decide to perform one. Initially we triggered a full collection when we had exhausted the currently allocated heap. The heap was not expanded unless a full collection had been unsuccessful. The full collection consisted of a partial collection followed by a parallel trace operation ***TR.*** The hope was that the partial collection would generally reclaim enough memory to let the mutator threads continue.

This approach has a number of problems. First, the heap is often exhausted by allocation of large blocks of storage. These often require completion of the next collection, and perhaps a heap expansion, before they can be satisfied. Even if this is not the case, there is no opportunity to run ***M*** before the partial collection without stalling the allocating thread for its duration. To make matters worse, the allocating thread may hold a crucial lock, thus also stalling other threads.

This lead us to a model in which the collector is triggered solely by a daemon thread, which watches how much allocation has taken place. Full collections are triggered if the amount of apparently live memory exceeds the amount of live memory at the end of the last collection by a certain amount. If a full collection is needed, a normal partial collection is started, including up to two iterations of ***M***. This is followed immediately by a completely concurrent execution of ***TR***. If the allocator ever exhausts memory, it tries to immediately expand the heap.

This policy is a bit dangerous, in that the heap may grow rapidly if the collector falls far behind. To reduce this danger we exercise control over the scheduling of the collector and mutator threads, such that the fraction of time allocated to the mutator drops off rapidly, but smoothly, as the collector falls behind.

Other policy decisions surround the question of which pages to use for allocation of small objects. We avoid allocation on a page that is already 3/4 full, so that we do not unnecessarily dirty it. It is unknown whether this is a good choice.

6. Empirical results

The mostly parallel generational collector described in the previous section has been in routine use on SPARCStations, as part of the Xerox Portable Common Runtime (PCR) and PCedar [Weiser 89], for several months. This paper was edited on a system that uses it.

The collector marking code has been quite heavily tuned and optimized. However, the same is not true for some other pieces of code run for our measurements. For example, allocation time (exclusive of collection) could have been reduced by about 50% by running a streamlined, less general, assembly coded allocator. (It could have been reduced still further if we were operating in a world in which there is no concurrency aside from the collector.)

We used the PCR preemptive thread-scheduling facility [Weiser 89] to allow the collector to run concurrently with the mutator. All measurements were performed such that all threads were run by a single UNIX process. A page fault thus stopped all threads. The code is written to allow more than one UNIX process to run threads, and has often been run in this mode (with slightly worse performance). Similarly, no fundamental changes would be needed if those UNIX processes were scheduled on more than one physical processor, provided UNIX shared memory across processors were supported. We did not address the question of running the collector on more than one processor simultaneously, though aside from the unlikely possibility of extremely deep and narrow linked data structures, this would not be terribly difficult to do.

The collector was implemented so as not to require modification to the vendor supplied operating system. Dirty bit information (on *virtual* memory pages) was thus not derived from the hardware dirty bits. Instead the entire heap was write protected. The resulting write faults were caught as UNIX signals at user level, and recorded. Various Portable Common Runtime interfaces to SunOS system calls were modified so as to preclude unrecoverable write faults in system calls. The primary cost of this is that the first time a page in the heap is written after a garbage collection, a signal must be caught and a system call must be executed to unprotect the page. The cost of this is variable, but in our environment appears to be somewhat less than half a millisecond per page written.

The allocator distinguishes between objects containing pointers and those known never to contain pointers. The implementation performs partial collections after allocating approximately one quarter as many bytes as there are in pointer-containing objects. (This heuristic is an attempt at bounding the fraction of time spent collecting. In our environment, collection time is very roughly proportional to the total size of pointer-containing objects.)

We are really interested in measuring interactive response in the presence of garbage collection. Subjectively, this improved substantially with the parallel generational collector. However, interactive sessions are difficult to reproduce and measure in different environments. Thus we resorted to running toy programs. But, since we are interested in the performance of the collector in a large single address space system, these toy programs are run in the same address space with the Cedar window system, the Tioga editor, a mailer, the SchemeXerox system [CurtisRauen 90], and a typical complement of miscellaneous smaller tools. These summed to roughly 70,000 objects, between 9.5 and 10 megabytes of pointer-free allocated objects and between 2.5 and 3 megabytes of pointer-containing allocated objects, in a 20 megabyte heap. Much of the pointer-free space is used for object code and static data for the Cedar/Mesa program implementing the environment. The static data areas are treated as roots by the collector.

We attempted to measure comparable stop-the-world full, generational, and parallel generational collectors. However, it is unfair to run the full collector more frequently than when the heap is exhausted. There is usually little to be gained from more frequent collections. But the other approaches benefit from more frequent collections. As a compromise, we fixed the heap size at 20 megabytes, ran the full collector only when the heap was full, triggered the other two collectors from a daemon thread, and tuned the parameters of the parallel/generational collector such that the heap size would remain at 20 megabytes. (The parallel generational collector running Boyer on the 10 MB machine did expand the heap to 21 MB near the end of the run.) This meant that the nonparallel generational collector ended up running more frequently than absolutely necessary, since it did not really need the reserve space for allocation during collection. Overall, this probably also increased its running time relative to the other two, but decreased pause times.

The two programs we consider here are five iterations of the Boyer benchmark, as compiled by SchemeXerox, and a simple allocator loop, written in C. The former is described in [Gabriel 85], and is often (ab)used as a garbage collector benchmark. The version of the SchemeXerox compiler we used was rather preliminary. Thus the absolute execution times are considerably longer than they should be. One cause for this is that cons-cells are 16 bytes long.

The latter program allocates two and a half million 8 byte objects and does not preserve any references to any of them. Both programs probably exhibit more local write behavior then the average interactive session, with the simple allocator presenting the extreme case. But real systems tend to be less allocation intensive than both benchmarks, creating less danger of the allocator falling behind. In our experience, real behavior tends to be close to that for the Boyer benchmark.

Each program was measured on a 36 Megabyte SPARCStation 2, and also on the same machine reconfigured to use only 10 MB of its memory. The machine had a local paging disk. The 36 MB machine does not page significantly, whereas the 10 MB runs of Boyer were completely disk bound, as can be seen from the time differences. Note that the thread stacks and PCR are separate from the heap, that the 10 MB figure also includes the UNIX kernel, that the Scheme system itself relies on substantial parts of the Cedar environment, and that a number of non-Scheme-related daemon threads run occasionally. All of these no doubt contributed to the paging behavior.

In each case we report total execution times, number of garbage collections (number of full collections in parentheses), and maximum and average pause times for garbage collections. Total execution times are given in seconds, while garbage collection pause times are given in milliseconds. Total execution times were reproducible to within about 10%. Pause time averages occasionally varied by up to 20%, but were usually also within 10% of each other.

10 MB, alloc20meg

	Total time	No. of colls.	Pause Times	
			Max.	Ave.
	secs	(full)	*msecs*	*msecs*
full	332.2	3(3)	51350	46323
gen	24.4	20(0)	870	125
gen,par	32.0	11(0)	350	102

10 MB, Boyer

	Total time	No. of coll.	Max.	Ave.
full	512.6	4(4)	63380	56548
gen	360.6	22(5)	41840	9291
gen,par	259.6	12(2)	329	169

36 MB, alloc20meg

	Total time	No. of coll.	Max.	Ave.
full	16.4	3(3)	1040	1037
gen	15.5	17(0)	200	81
gen,par	17.2	11(0)	100	76

36 MB, Boyer

	Total time	No. of coll.	Max.	Ave.
full	51.6	4(4)	1610	1368
gen	60.4	22(5)	1471	528
gen,par	66.1	13(2)	159	134

On a machine configured for 8 MB memory, even the parallel collector pause times went up to an average of about 2 seconds, and the heap grew by 2 megabytes, indicating that some parameters needed to be tuned to keep the mutator from getting too far ahead during full collections.

The average pause time for the generational collector running Boyer on the small memory machine is completely dominated by the full collection times. The smaller partial collection times were around 300 milliseconds. This effect does not arise for the allocator loop, since it requires no full collections.

Mostly parallel collection significantly reduces average pause times below those attainable by pure generational collection, both by avoiding pauses associated with full collections, and by reducing pause times for partial collections. (This is of course less true for the allocator loop, since it touches only the pages it allocates from, and requires no full collections, thus making the page cleaning operations less productive. But note again that parallel collection were effectively less frequent, since a lot of allocation occurred during each collection. Thus total pause times were significantly less.) The benefit of hiding full collections is most noticeable on small memory machines, where a full collection requires large amounts of paging. On memory-rich machines, even full stop-the-world collection times can be sufficiently short that they are only a minor annoyance.

In our environment collection pauses are usually unnoticeable (unlike network related pauses). Short pauses are achieved at moderate additional cost in processor time, since roots and dirty objects may be scanned repeatedly.

Note that PCedar itself (i.e. the mutator) is barely usable on a 10 megabyte machine. Nonetheless the parallel collector keeps collection pauses tolerable. They are significantly shorter than response time for a command that has not been run recently, and has thus had some of its code paged out.

The number of full collections observed during the experiments indicates that a relatively large number of short-lived objects are getting marked, and thus surviving to the next full collection. Since full collections are usually not disastrous, this problem can be tolerated. However, we are exploring modifications to the generational collection scheme similar to those in [DemersEtAl 90] that would improve this behavior without adding significantly to the required bookkeeping overhead.

7. Mostly Parallel Copying Collectors

It is possible to apply the same approach to obtain a mostly parallel copying collector. Unlike the [AppelEllisLi 88] collector, this approach requires only dirty bit information. Unfortunately, it also appears to require additional space to maintain explicit forwarding links. We assume that every object has an additional field called *forward*, which is set and examined only by the garbage collector. The underlying collection algorithm can be either traditional breadth-first copying (cf. [Cheney 70]) or one that attempts to preserve better locality of reference (cf. [Moon 84]).

The copying collector is invoked concurrently with the mutator. As usual, the collector copies all reachable objects residing in *from-space* to a previously unused region of memory referred to as *to-space*. Links in *to-space* are updated to reflect the new locations of the objects. This concurrent collector is identical to the sequential version, except in that

1) It clears the *forward* pointers in *from-space* before it starts. (We assume that the collector can write *forward* fields without affecting dirty bits. This may require allocating *forward* pointers separately, *e.g.* in a part of *to-space* that will not be immediately needed.)

2) It maintains information about pages dirtied since the beginning of the collection.

3) It stores the new address of each copied object into the forwarding link of each object in *from-space*.

The mutator continues to see only *from-space* objects. (In the [AppelEllisLi 88] collector, the mutator sees only *to-space* objects.)

This concurrent collection process establishes the condition that if an object residing on a clean page has been copied, then every object it points to has also been copied. Furthermore, if an object resides on a clean page then its copy has the correct contents. With the world stopped, we can then run following finishing operation to ensure that all reachable objects have been copied, and all copies contain the correct contents:

$\boldsymbol{F_c}$: For every copied object *a* whose *from-space* copy resides on a dirty page:

1) Copy any objects that the *from-space* copy of *a* points to, that have not yet been copied, i.e. that have *NIL* forwarding links.

2) Update pointers in copies to refer to *to-space*, recursively copying uncopied objects. (This can be done without a stack, as with the original copying collector. Breadth first copying is probably fine here, since this should be a small collection of objects that have all been referenced within a short time interval.)

3) Recopy *a* to reflect changes in both pointer and nonpointer fields that occurred since the start of the collection.

As in the noncopying collector, it is easy to construct a variant of $\boldsymbol{F_c}$ that can be run concurrently to further reduce the amount of time expended by the final stop-the-world collection. Again, the amount of time spent with the world stopped is proportional to the number of pages dirtied since the start of the last parallel copying phase, and thus should be quite short.

We have not built such a collector, since it is not practical in our environment. An empirical performance comparison with the [AppelEllisLi 88] collector would be interesting. Our alternative is most likely to be attractive if the operating system provides inexpensive dirty bit access, but relatively expensive trap handling.

Acknowledgements

Bob Hagmann and Barry Hayes suggested some of the alternatives described in section 5. UNIX is a trademark of AT&T Bell Laboratories. SPARCStation is a trademark of Sun Microsystems.

References

[AppelEllisLi 88] Appel, Andrew, John R. Ellis, and Kai Li, ‘‘Real-time Concurrent Collection on Stock Multiprocessors’’, *Proceedings of the SIGPLAN ’88 Conference on Programming Language Design and Implementation, SIGPLAN Notices 23*, 7 (July 88), pp. 11-20.

[AppelLi 91] Appel, Andrew W., and Kai Li, ‘‘Virtual Memory Primitives for User Programs’’, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[Bartlett 89] Bartlett, Joel F., ‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++’’, DEC WRL Technical Note TN-12, October 1989.

[BoehmWeiser 88] Boehm, Hans-J. and Mark Weiser, ‘‘Garbage Collection in an Uncooperative Environment’’, *Software Practice & Experience 18*, 9 (Sept. 1988), pp. 807-820.

[Cheney 70] Cheney, C., J., ‘‘A Nonrecursive List Compacting Algorithm’’, *Communications of the ACM 13,* 11 (November 1970), pp. 677-678.

[CurtisRauen 90] Curtis, P. and J. Rauen. A Module System for Scheme. *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, June 1990, pp. 13-19.

[DemersEtAl 90] Demers. A., M. Weiser, B. Hayes, H. Boehm, D. Bobrow, S. Shenker, ‘‘Combining Generational and Conservative Garbage Collection: Framework and Implementations’’, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, January 1990, pp. 261-269.

[DeTreville 90] DeTreville, John, ‘‘Experience with

Concurrent Garbage Collectors for Modula-2+'', Digital Equipment Corporation, Systems Research Center, Report No. 64.

[DikstraEtAl 78] Dijkstra, E. W., L. Lamport, A. Martin, C. Scholten, and E. Steffens, ''On-the-Fly Garbage Collection: An Exercise in Cooperation'', *Communications of the ACM 21*, 11 (November 78), pp. 966-975.

[Gabriel 85] Gabriel, Richard P., *Performance and Evaluation of Lisp Systems*, MIT Press, 1985.

[Moon 84] Moon, D., ''Garbage Collection in Large Lisp Systems'', *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 235-246.

[Rovner 84] Rovner, Paul, ''On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically Checked, Concurrent Language'', Report CSL-84-7, Xerox Palo Alto Research Center.

[Steele 75] Steele, Guy L., ''Multiprocessing Compactifying Garbage Collection'', *Communications of the ACM 18*, 9 (September 75), pp. 495-508.

[Ungar 84] Ungar, David, ''Generation Scavenging: a non-disruptive high performance storage reclamation algorithm'', *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices 19*, 5 (1984), pp. 157-167.

[Weiser 89] Weiser, M., A. Demers, and C. Hauser, ''The Portable Common Runtime Approach to Interoperability'', *Proceedings of the 13th ACM Symposium on Operating System Principles* (December 1989).

[Wentworth 90] Wentworth, E. P., ''Pitfalls of Conservative Garbage Collection'', *Software Practice & Experience 20*, 7 (July 1990) pp. 719-727.

[Zorn 90] Zorn, Benjamin, ''Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection'', *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990, pp. 87-98.