Combining Generational and Conservative Garbage Collection: Framework and Implementations

Alan Demers
Mark Weiser
Barry Hayes
Hans Boehm
Daniel Bobrow
Scott Shenker

Xerox Palo Alto Research Center
Palo Alto, Ca 94304

SUMMARY

Two key ideas in garbage collection are *generational collection* and *conservative pointer-finding*. Generational collection and conservative pointer-finding are hard to use together, because generational collection is usually expressed in terms of copying objects, while conservative pointer-finding precludes copying. We present a new framework for defining garbage collectors. When applied to generational collection, it generalizes the notion of younger/older to a partial order. It can describe traditional generational and conservative techniques, and lends itself to combining different techniques in novel ways. We study in particular two new garbage collectors inspired by this framework. Both these collectors use conservative pointer-finding. The first one is based on a rewrite of an existing trace-and-sweep collector to use one level of generation. The second one has a single parameter, which controls how objects are partitioned into generations; the value of this parameter can be changed dynamically with no overhead. We have implemented both collectors and present measurements of their performance in practice.

This appeared in the Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages, January 17-19, 1990, pp. 261-269.

I. Introduction

Garbage collectors fall into two general classes: reference-counting and tracing. In this paper we consider only tracing collectors. A tracing garbage collector works by starting with a root set of memory objects, following the pointers found there to other memory objects that should be preserved, and so on recursively, until all objects accessible from the roots have been found. Inaccessible objects are garbage and can be reclaimed.

Garbage collection has a colorful past. It is considered essential by some programming subcultures, such as those from a Lisp heritage, and is considered superfluous or dangerous by other subcultures, such as those from a systems programming or real-time background. However, there has been some use of garbage collection in systems programming [Rovner85] [Weiser89] [Cardelli88], and even real-time constraints are possible [Baker78] [Appel88]. In general, interest in garbage collection is growing.

Garbage collection is almost never shared among multiple language implementations. Instead, every language with garbage collection does it differently, even idiosyncratically, because collection usually depends on implementation assumptions about the uses of pointers. There is, however, a technique for identifying pointers that is nearly language-independent. This technique, called *conservative pointer-finding*, identifies a superset of the true pointers, in effect by treating every word of a memory object as if it *might* possibly contain a pointer [Boehm88]. Conservative pointer-finding precludes copying objects when an object is copied, every pointer to that object must be updated to refer to the new location, but conservative pointer-finding cannot distinguish pointers from integers, so updating them is not safe.

One can combine partially conservative with partially copying collection [Bartlett88]. Other techniques, such as calling language-dependent pointer-finding routines during the trace (which we do), further relieve the problems of conservative pointer knowledge.

A key method for achieving high performance garbage collection is to concentrate reclamation effort on recently allocated objects. This technique, a version of which was used as early as 1975 in the SITBOL collector [Hanson77] [Ripley78], has come to be known as "generational collection", because it classifies objects by how old they are [Lieberman83]. Generational collectors have always been copying collectors, partly because copying results in more compact storage (and thus fewer total pages in use), but primarily because generations have always been defined and implemented in terms of different memory spaces segregated by object ages. Since generational collection has become widely known, almost all new implementations use this technique [Unger84] [Moon84] [Sobalvarro88] [Courts87] [Wilson89]. It has substantial performance benefits, and is compatible with real-time performance requirements.

Conservative collection has great potential for being the foundation of a language-independent system of collection, but precludes copying in the general case. Generational collection is extremely important for high-performance collection, but seems to require copying of objects. Combining the two is the challenge we took on, and the result is two-fold: first, a better theoretical framework for understanding garbage collection in general, and generational collection in particular; second, several new implementations of garbage collectors based on this theoretical framework. Although our framework applies to previous generational collectors as well, the new ones we have implemented all have the property that they use conservative pointers and so do not copy. Partially copying collectors, in the style of Bartlett, are also possible, but we haven't done them.

II. Collector Theory

Storage model and partial garbage collection

We fix on a model of computer storage in which there is a countably infinite set O of *objects*, with a distinguished *root object* $\mathbf{r} \in O$.

1: Definition. A *storage state S* is a pair $<AS, P>$, where AS, the *allocated set,* is a finite subset of O, and $P \subseteq O \times O$, the *points-to relation,* is a reflexive binary relation on O. ♦

2: Definition. A storage state is *valid* if it satisfies the following invariants:

$I_0 : \mathbf{r} \in AS$.

$I_1 : ((<a, b> \in P) \wedge (a \in AS)) \Rightarrow (b \in AS)$.

$I_2 : ((<a, b> \in P) \wedge (a \notin AS)) \Rightarrow (b = a)$. ♦

Reflexivity of P is required for technical reasons. Since the garbage collectors described below are essentially computing the reflexive transitive closure P^*, the assumption that P is reflexive has no effect on their operation.

3: Remark. In any valid storage state, all objects reachable from the root are allocated; that is, $P^*(\mathbf{r}) \subseteq AS$. ♦

The client program, or *mutator*, repeatedly makes changes to the storage state. The only assumption we make about these changes is that they preserve validity.

The *garbage collector*, which runs repeatedly and atomically with respect to the mutator, *reclaims* selected objects by deleting them from AS; the collector is *valid* if it accomplishes this without changing P on allocated objects and without violating the invariants. Formally,

4: Definition. A *garbage collection* of storage state $<AS, P>$ is a a storage state $<AS', P'>$ such that

$AS' \subseteq AS$, and

$(a \in AS') \Rightarrow (P'(a) = P(a))$.

The collection is *valid* if

$(<AS, P>$ valid $) \Rightarrow (<AS', P'>$ valid $)$.

A *(valid) garbage collector* is a function mapping storage states to (valid) garbage collections. ♦

Stated in these terms, a garbage collector is completely characterized by the strategy it uses to choose AS', the set of *retained* objects. Clearly, choosing AS' to be exactly the *reachable* set $P^*(\mathbf{r})$ yields a valid collection. We call this collection *precise*, because of the following:

5: Lemma. If $<AS', P'>$ is a valid garbage collection of $<AS, P>$, then

$AS' = P^*(AS') \supseteq P^*(\mathbf{r})$. ♦

A collection that retains a proper superset of $P^*(\mathbf{r})$ is called *partial* (since it reclaims only part of the unreachable memory). As we discuss below, there can be compelling practical reasons for building a valid partial garbage collector rather than a precise one.

Not every superset of $P^*(\mathbf{r})$ yields a valid partial collection according to our theory. By Lemma 5 above, we require that the retained set AS' be closed under P. It is possible to argue that *every* tracing collector must identify a "reachable" set that is closed under P. However, one can imagine building a collector that does not reclaim all the objects it identifies as unreachable (e.g. because some of them are on nonresident pages). Our theory would not apply to such a collector. In addition, the theory says nothing about running time, which is in any event implementation-dependent. The theory does enable us to determine formally sets of objects that can be collected and sets of objects that may safely be ignored during a collection.

We can characterize those supersets of $P^*(\mathbf{r})$ that yield valid partial collections with the help of the following:

6: Definition. Let $<AS, P>$ and $<AS, Q>$ be valid storage states. Then $<AS, Q>$ is a *pointer augmentation* of $<AS, P>$ if $Q \supseteq P$. ♦

7: Theorem. Let $<AS, P>$ be a valid storage state. $AS' \subseteq AS$ yields a valid garbage collection iff $AS' = Q^*(\mathbf{r})$, where $<AS, Q>$ is a pointer augmentation of $<AS, P>$. ♦

This theorem states that, for the class of garbage collectors we are considering, every valid partial collection is equivalent to a precise collection on a pointer augmentation of the true state.

Posets, embeddings and pointer augmentation

Here we define an embedding of a storage state in a partially-ordered set. We argue that any pointer augmentation can be induced by an appropriately-chosen embedding. By Theorem 7 above, every valid partial collection is equivalent to a pointer augmentation plus a precise collection. Combining these results, we are assured that one can fully explore the space of valid partial garbage collections by exploring the space of embeddings.

8: Definition. A *pointed partially ordered set*, (hereafter *poset*), is a triple $<D, \geq, \perp>$, where D is a nonempty set, $\geq$ is a reflexive, transitive and antisymmetric relation on D, and $\perp \in D$ such that $\forall x{\in}D\ x{\geq}\perp$. ♦

9: Definition. Let $\boldsymbol{D} = <D, \geq, \perp>$ be a poset, and let $S = <AS, P>$ be a valid storage state. An *embedding* of S into $\boldsymbol{D}$ is a pair $<F, A>$ of functions from O into D such that

$$\forall\ a, b \in O\ \ <a, b> \in P \Rightarrow F(a) \geq A(b).$$

An embedding $<F, A>$ determines an *induced points-to relation* $P_{F,A}$ on O by

$$<a, b> \in P_{F,A}\ \ iff\ \ F(a) \geq A(b) \wedge ((a = b) \vee ((a \in AS) \wedge (b \in AS))).$$

The embedding is said to be *lossless* if $P_{F,A} = P$. ♦

A natural correspondence between embeddings and pointer augmentations is given by the following two lemmas. Lemma 10 states that every embedding induces a pointer augmentation; lemma 11 states that every pointer augmentation is induced by some embedding.

10: Lemma. Let $S = <AS, P>$ be a valid storage state, and let $<F, A>$ be an embedding of S in poset $\boldsymbol{D} = <D, \geq, \perp>$. Then $<AS, P_{F,A}>$ is a pointer augmentation of $<AS, P>$, and $<F, A>$ is a lossless embedding of $<AS, P_{F,A}>$ in $\boldsymbol{D}$. ♦

11: Lemma. Let $S = <AS, P>$ be a valid storage state, and let $S' = <AS, P'>$ be a pointer augmentation of S. Then there exists a poset $\boldsymbol{D} = <D, \geq_{\boldsymbol{D}}, \perp_{\boldsymbol{D}}>$ and an embedding $<F, A>$ of S in $\boldsymbol{D}$ such that $P_{F,A} = P'$. That is, $<F, A>$ is a lossless embedding of S' in $\boldsymbol{D}$. ♦

Pointer augmentations can be performed directly on embeddings with the aid of the following:

12: Definition. Let $\boldsymbol{D} = <D, \geq_{\boldsymbol{D}}, \perp_{\boldsymbol{D}}>$ and $\boldsymbol{E} = <E, \geq_{\boldsymbol{E}}, \perp_{\boldsymbol{E}}>$ be posets. A function h from D to E is a *homomorphism* from $\boldsymbol{D}$ to $\boldsymbol{E}$ if it is strict and monotonic. ♦

13: Lemma. Let $\boldsymbol{D} = <D, \geq_{\boldsymbol{D}}, \perp_{\boldsymbol{D}}>$ and $\boldsymbol{E} = <E, \geq_{\boldsymbol{E}}, \perp_{\boldsymbol{E}}>$ be posets, and let h be a homomorphism from $\boldsymbol{D}$ to $\boldsymbol{E}$. Let $S = <AS, P>$ be a valid storage state, and $<F, A>$ an embedding of S in $\boldsymbol{D}$. Then $<h{\circ}F, h{\circ}A>$ is an embedding of S in $\boldsymbol{E}$ with $P_{h{\circ}F,h{\circ}A} \supseteq P_{F,A} \supseteq P$. ♦

In the above lemma, $<AS, P_{h{\circ}F,h{\circ}A}>$ is a pointer augmentation of $<AS, P>$. Further, $<AS, P_{h{\circ}F,h{\circ}A}>$ is losslessly embedded in $\boldsymbol{E}$ by $<h{\circ}F, h{\circ}A>$. Thus, a homomorphism on the range of an embedding induces a pointer augmentation. As before, we can prove something very close to the converse:

14: Lemma. Let $S = <AS, P>$ be a valid storage state. There exists a poset $\boldsymbol{D} = <D, \geq_{\boldsymbol{D}}, \perp_{\boldsymbol{D}}>$, called a *canonical* poset for S, and a lossless embedding $<F, A>$ of S in $\boldsymbol{D}$, called a *canonical* embedding for S, with the following property. For any pointer augmentation $S' = <AS, P'>$, there exists a poset $\boldsymbol{D}' = <D', \geq_{\boldsymbol{D}'}, \perp_{\boldsymbol{D}'}>$ and a homomorphism h from $\boldsymbol{D}$ to $\boldsymbol{D}'$ such that $<h{\circ}F, h{\circ}A>$ is a lossless embedding of S' in $\boldsymbol{D}'$.

Proof (sketch): To construct a canonical embedding for storage state $S = <AS, P>$, we let

$$D = \{\perp_{\boldsymbol{D}}\} \cup (\ O \times \{0, 1\}\ ),$$

and let $\geq_{\boldsymbol{D}}$ be the reflexive closure of the relation

$$\{ <<a, 0>, <b, 1>> \mid <a, b> \in P \}.$$

It is easy to verify that $\geq_{\boldsymbol{D}}$ so defined is transitive and antisymmetric, so $\boldsymbol{D} = <D, \geq_{\boldsymbol{D}}, \perp_{\boldsymbol{D}}>$ is a poset. It is also easy to verify that the pair of functions $<F, A>$ given by

$$F(a) = <a, 0> \qquad A(a) = <a, 1>$$

is a lossless embedding of S in $\boldsymbol{D}$. Now if $S' = <AS, P'>$ is any pointer augmentation of S, the identity function on $(O{\times}\{0, 1\})$ yields the required homomorphism between the canonical posets for S and S'. ♦

Thus, any pointer augmentation on a storage state S can be induced by a suitably chosen homomorphism on the canonical embedding for S.

Constructing partial collectors from embeddings

In practice, there are two basic reasons to build a partial collector rather than a precise one. First, the actual points-to relation may be unavailable. This comes about in languages like C, where the typing system provides too little information to identify all the pointers, making it necessary to use a conservative pointer-finding strategy. Second, some easily-identified subset of the allocated objects may be richer in collectable objects than the entire allocated set. In that case, collecting from only that subset can reclaim a large fraction of the unreachable objects at substantially less cost than a full collection. This is the basis of generational collection schemes. The savings that result can be dramatic on machines with virtual memory a well-designed generational collector can cause far fewer page faults than a full collector. Either of these sources of imprecision, or their combination, can be expressed in a natural way as the pointer augmentation induced by an embedding.

It is straightforward to express conservative pointer-finding: the conservative points-to relation is simply a pointer augmentation of the true points-to relation, and so by Lemma 11 can be expressed as the induced points-to relation of an embedding. Our storage model disallows pointers to unallocated objects, and the conservative pointer-finding collectors described below do the same roughly, a bit pattern is not treated as a pointer unless its value is the address of an allocated object.

Describing a generational garbage collector is only slightly more involved. Informally, a generational collector works by partitioning the allocated objects into *threatened* objects, which are candidates for collection, and *immune* objects, which will not be collected. Optionally, the collector may then be able to identify efficiently a set of objects called the *bystanders*, which are guaranteed not to contain pointers into the threatened set. The identified bystander set need not include *all* objects without pointers into the threatened set, but it is advantageous to identify as many bystanders as possible, since the collector does not need to trace through bystanders.

For a garbage collection described by an embedding, immune and bystander sets are easily identified as follows. Let $S = <AS, P>$ be a valid storage state, let $\boldsymbol{D} = <D, \geq, \perp>$ be a poset, and let $<F, A>$ be an embedding of S in $\boldsymbol{D}$. Consider the collection described by $<F, A>$, that is, the precise collection on $<AS, P_{F,A}>$. The immune set consists of all allocated objects in $A^{-1}(\perp)$. Similarly, the bystander set consists of all allocated objects in $F^{-1}(\perp)$. To see this, consider the points-to relation $P_{F,A}$. The pairs in this relation that are of interest to us are the ones between allocated objects, since these are the only ones that affect the reachable set. By Definition 9, if a and b are allocated objects and $b \in A^{-1}(\perp)$, then $<a, b> \in P_{F,A}$. That is, every allocated object in $A^{-1}(\perp)$ is pointed to by every allocated object. In particular, every allocated object in $A^{-1}(\perp)$ is directly pointed to by the root object **r**. Thus, no such object can be collected, and we can identify $AS \cap A^{-1}(\perp)$ as the immune set. Again by Definition 9, if $a \in F^{-1}(\perp)$ and $<a, b> \in P_{F,A}$ then $b \in A^{-1}(\perp)$. That is, the only nontrivial pointers from objects in $F^{-1}(\perp)$ go to allocated objects in $A^{-1}(\perp)$. Since there are no pointers from objects in $F^{-1}(\perp)$ to objects in the threatened set (which is given by $AS - A^{-1}(\perp)$), we can identify $AS \cap F^{-1}(\perp)$ as the bystander set.

To describe generational garbage collection, we would like to be able first to choose an immune set and then to introduce a pointer augmentation that determines exactly the chosen immune set. Here we describe a useful family of immune sets for which this process is particularly easy.

15: Definition. An *ideal* in a poset $\boldsymbol{D} = <D, \geq, \perp>$ is a nonempty downward-closed subset of D. ♦

As before, we begin with an embedding $<F, A>$ of storage state S in poset $\boldsymbol{D}$. We choose an ideal $I \subseteq D$ such that $AS \cap A^{-1}(I)$ is the desired immune set. Define the function h from D to D by

$$h(x) = \begin{cases} \perp & x \in I \\ x & \text{otherwise} \end{cases}$$

Clearly h yields a homomorphism from $\boldsymbol{D}$ to $\boldsymbol{D}$. Thus, by Lemma 13, $<AS, P_{h°F,h°A}>$ is a pointer augmentation of $<AS, P_{F,A}>$, and $<h°F, h°A>$ is a lossless embedding of $<AS, P_{h°F,h°A}>$ into $\boldsymbol{D}$. Straightforward computation yields

$$P_{h°F,h°A} = P_{F,A} \cup \{ <a,b> \mid a \in AS \wedge b \in AS \wedge b \in (h°A)^{-1}(\perp) \}.$$

That is, $P_{h°F,h°A}$ consists exactly of $P_{F,A}$ augmented by pointers from all allocated objects to the objects in the chosen immune set. A precise garbage collection of $<AS, P_{h°F,h°A}>$ is the best one possible, given the choice of immune set, in the following sense: for any valid collection $<AS', P'>$ of $<AS, P_{F,A}>$,

$$AS' \supseteq (AS \cap A^{-1}(I)) \Rightarrow AS' \supseteq (P_{h°F,h°A})^{*}(\mathbf{r}).$$

That is, any valid collection of $<AS, P_{F,A}>$ that retains all the chosen immune objects must retain all the objects in $(P_{h°F,h°A})^{*}(\mathbf{r})$.

Combining collection strategies

Here we show how the collections described by two different embeddings can be combined in a natural way by combining the embeddings.

16: Definition. Let $\boldsymbol{D}_1 = <D_1, \leq_1, \perp_1>$ and $\boldsymbol{D}_2 = <D_2, \leq_2, \perp_2>$ be posets. The *(strict) Cartesian product* $\boldsymbol{D}_1 \times \boldsymbol{D}_2$ is the poset $\boldsymbol{D} = <D, \leq, \perp>$ where

$$D = \{\perp\} \cup \{ <x_1, x_2> \mid x_1 \in D_1 - \{\perp_1\} \wedge x_2 \in D_2 - \{\perp_2\} \},$$
$$\perp \leq \perp,$$
$$\perp \leq <x_1, x_2>, \text{ and}$$
$$<x_1, x_2> \leq <y_1, y_2> \text{ iff } x_1 \leq_1 y_1 \wedge x_2 \leq_2 y_2.$$

Below we use the notational convention that pairing is a strict operation; i.e., that

$$<x_1, \perp> = <\perp, x_2> = <\perp, \perp> = \perp.$$ ♦

17: Definition. Let $f : S \rightarrow D$ and $g : S \rightarrow E$ be functions; we define $f \otimes g : S \rightarrow D \times E$ by

$$(f \otimes g)(x) = <f(x), g(x)> \quad \forall x \in S.$$ ♦

18: Remark. If $f(x) = \perp$ or $g(x) = \perp$ then $(f \otimes g)(x) = \perp$. In particular,

$$(f \otimes g)^{-1}(\perp) = f^{-1}(\perp) \cup g^{-1}(\perp).$$ ♦

19: Lemma. Let $S = <AS, P>$ be a valid storage state; let $<F, A>$ and $<F', A'>$ be embeddings of S into $\boldsymbol{D}$ and $\boldsymbol{D'}$, respectively. Then $<F \otimes F', A \otimes A'>$ is an embedding of S into $\boldsymbol{D} \times \boldsymbol{D'}$. Further,

$$P_{F \otimes F', A \otimes A'} = P_{F,A} \cap P_{F',A'} \cup \{ <a, b> \mid (a \in AS) \wedge (b \in AS) \wedge ((A \otimes A')(b) = \perp) \}.$$ ♦

The last term in the expression for $P_{F \otimes F', A \otimes A'}$ above arises from strictness of the pairing operation. Intuitively, $P_{F \otimes F', A \otimes A'}$ consists of $P_{F,A} \cap P_{F',A'}$ augmented by pointers from all allocated objects to the objects in the union of the immune sets determined by A and by A'. Thus, a precise collection of $<AS, P_{F \otimes F', A \otimes A'}>$ has the effect of using the union of the immune sets together with all the pointer information available from $P_{F,A}$ and $P_{F',A'}$.

Taking the Cartesian product of two embeddings in this way is a particularly powerful tool for defining a generational collector. For example, let $<F, A>$ be an embedding of S into poset $\boldsymbol{D}$ that induces the most accurate points-to information available. Without loss of generality, assume that $A^{-1}(\perp) = \varnothing$. (It is easy to exclude $\perp$ from the range of A, by adding a new bottom element and "lifting" $\boldsymbol{D}$ if necessary). Let $<F', A'>$ be an embedding of S into $\boldsymbol{D'}$ such that $(AS \cap A'^{-1}(\perp))$ is the desired immune set. Consider the precise collection of $<AS, P_{F \otimes F', A \otimes A'}>$. For all threatened objects the induced points-to relation $P_{F \otimes F', A \otimes A'}$ is at least as accurate as $P_{F,A}$. The immune and threatened sets, however, are determined entirely by $P_{F',A'}$. Thus, we can make $<F', A'>$ as simple (and as easy to compute) as we desire without losing any of the pointer information encoded in$<F, A>$.

III. Introduction to Practice

We have implemented two collectors described by the theory presented above. For each collector, we first explain its operation in terms of an intuitive partial order and *A* and *F* functions. We then summarize the implementation and present some performance numbers to indicate that the theory leads to practical new strategies.

Both our implementations use total orders, rather than strictly partial orders. We believe there are important uses of partial orders, but our first order of business was to show that our theory led to new and practical collectors in more conventional domains, and this led us to generational collection, where the natural partial order is linear time.

Both our implementations use a trick of implementation of *A*'s and *F*'s motivated by temporal causality of pointers, as follows: A pointer written at time t cannot point to an object allocated at time $t' > t$. Therefore, if $F(a)$ is the time at which a last had a pointer written to it, and $A(b)$ is the time of allocation of b, then the invariant

$$\forall\, a, b \in O \;\; <a, b> \in P \Rightarrow F(a) \geq A(b)$$

is true by causality. Naturally this is not as precise as possible, since the pointer written at time $F(a)$ might have been to an object with allocation time well before $F(a)$. On the other hand, standard virtual memory support, such as page write-protection or dirty bits, is sufficient to enable us to maintain the invariant.

Another practical issue in our implementations is using summary, per-"card", information rather than per-object information. A card is a single contiguous region of memory together with information about the objects contained in that region. There are usually several cards per physical page of memory. Cards are used to reduce the per-object overhead of information like *A* and *F*, by storing the information only once per card. We use the term "card pollution" to refer to the imprecision that results from maintaining information on a per-card rather than a per-object basis. For example, when keeping *A* and *F* values per card, the invariant $\forall\, a, b \in O \;\; <a, b> \in P \Rightarrow F(a) \geq A(b)$ requires that $A(c)$ be the minimum *A*, and $F(c)$ be the maximum *F*, for all the objects on card c. Pollution then results as $A(c)$ gets smaller than max($A(a)$) for all a on c, and as $F(c)$ gets larger than min($F(a)$) for all a on c.

One basic technique to avoid pollution is to remove cards from use by the allocator when their objects have lasted for a few generations. This keeps max(*A*) from growing. It also has an indirect effect on *F* pollution by avoiding turnover of objects on the card, and so keeping the *F* values stable and not growing to point to objects with later birthdays. Each collector below has its own scheme for avoiding pollution.

IV. Collector I

As one experiment, we modified an existing trace-and-sweep collector (a descendant of the one described in [Boehm88]) to be generational. The strategy for a partial collection can be described in terms of the following embedding of the storage state. Consider the poset $B = <\{0,1\}, \geq, 0>$, with the intuition that 0 represents "old" objects, allocated before the last collection, and 1 represents "new" objects. We define $A(a)=1$ if a was allocated since the last collection, and 0 otherwise; $F(a)=0$ if it is known that a has not been altered since the last collection, and 1 otherwise. By the causality argument above, this definition of *A*'s and *F*'s preserves our invariants, and so is a legitimate embedding. Poset *B* has only one interesting ideal, namely {0}. As discussed in Section II above, the immune set associated with this embedding, $A^{-1}(0)$, consists of all objects allocated before the last collection.

The actual collection strategy used by Collector I is described by the Cartesian product of the above embedding with an embedding for conservative pointer-finding. We now show that this strategy allows a simple and efficient implementation.

Sticky Mark Bits

Collector 1 may be viewed as a modification of a conventional mark-sweep collector, in which we sometimes neglect to reset the mark bits between collections. In this way, every object that survived the last collection has its mark bit set, and thus the mark bit can also be interpreted as the *A* value of the object corresponding to the poset *B*. Therefore we dubbed this approach "sticky mark bit" collection.

The algorithm for a full garbage collection in a conventional mark-sweep collector can be expressed as follows:

1. Clear all mark bits.
2. Mark all objects reachable from the roots.
3. Reclaim all unmarked objects.

The algorithm for a partial garbage collection is only slightly different. We attempt to reclaim only those objects allocated since the last collection (i.e., not marked by it). Thus, for partial collections, we do not perform step 1. This implicitly establishes the *A* values corresponding to the poset *B*. The bystanders are those objects with *F* value of 0, that is, those objects that have not been altered since the last collection. These do not need to be considered. We assume that all other immune objects are reachable, and treat then as additional roots. This is implemented by replacing step 1 by:

1'. Mark from all modified marked objects.

As in Collector II below, we have no way of identifying modified objects other than through the paging hardware. Thus step 1' must be implemented as:

1''. Mark from all marked objects on dirty cards.

This is the only substantial difference between full and partial collections. In practice it is important to intersperse partial collections with full collections, since a significant number of short-lived objects will survive a single collection.

We can state A and F more precisely now in terms of our implementation: Every object has two properties: $M(a)$ is true if a is marked, and $D(a)$ is true if a is on a page that has been dirtied. Then the following mapping expresses A and F values in terms of the information maintained by the algorithm:

$$M(a) \Rightarrow A(a) = 0$$
$$\sim M(a) \Rightarrow A(a) = 1$$
$$M(a) \wedge D(a) \Rightarrow F(a) = 1$$
$$M(a) \wedge \sim D(a) \Rightarrow F(a) = 0$$
$$\sim M(a) \Rightarrow F(a) = 1$$

Results from Collector I

We replaced the standard garbage collector and allocator in Ibuki Common Lisp [IBUKI87] with our Collector I, using a card size of 4096 bytes. We measured collection times for the Boyer benchmark from the Gabriel benchmarks [Gabriel85], and for the Ibuki Common Lisp compiler compiling its two largest modules of about 1000 lines each. The heap size was fixed at 2.5M, and full trace-and-sweep was performed just before measurement began. The tests were performed on a Sun-3/260 with 24MB ram. All programs fit in real memory, and the machine was essentially unloaded during the tests. Measured times are in seconds of Unix user+system time.

We compared collection times for two different collection policies:

Policy 1: All collections are full trace-and-sweep, triggered when the heap is full.

Policy 2: A partial collection is triggered after approximately every 100 Kbytes of allocation. Such collections ignore cards that are more than 3/4 full. A full collection is triggered when all cards are more than 3/4 full.

Under both policies, the sweep phase is deferred almost entirely until allocation time. This may save some time, since some cards are never swept, and it always reduces garbage collection pauses. Dirty bits are simulated by checksumming cards. We excluded the deferred sweep time and the (substantial) checksumming overhead from the measurements.

For the Boyer benchmark under policy 1, a typical run took 3 collections, with a total time of 8.5 seconds. (About 2.7 additional seconds were spent sweeping during allocation.) Essentially the full heap of more than 600 pages was touched during every collection.

Under policy 2 there were typically 30 partial collections, plus an average of 1.4 full collections. An average of only 42 cards were touched per partial collection. Total collection time per iteration was 21.5 seconds (plus about 3.2 seconds delayed sweep overhead). The new collector was therefore slower in cpu time, but it touched far fewer pages during most collections. For applications running in limited physical memory, this advantage would certainly outweigh the increase in cpu time. Also, garbage collection pauses were reduced from about 3 seconds (under policy 1) to about 1/2 second, and thus would have been unnoticeable on a slightly faster machine.

The number of remaining full collections is relatively high in this example, since the heap becomes close to full. One explanation is that the heap is sufficiently full that even a complete collection may free only a third of the heap. We would also no doubt benefit from collecting slightly less frequently, thus reducing the number of short-lived objects that "accidentally" survive until the next full collection. This would also substantially reduce the required cpu time. We observed that if we also postpone partial collections until the heap fills up, we incur only about a 40% cpu time overhead, but still keep the number of pages touched in a partial collection down to about half.

For the compilation benchmark we obtained similar performance results. Policy 1 resulted in an average of 2.8 collections per benchmark iteration, with a garbage collection time of 7.3 seconds. Policy 2 reduced the number of full collections to exactly one per iteration, but added 26 partial collections. Total collection time went up to 20.7 seconds. About 57 cards were touched by each partial collection.

V. Collector II.

Our second collector, rather than using a single bit for time, uses the integers starting at zero. CurrentTime increases by one each collection. Rather than implicit A and F functions as in Collector I, we use explicit A and F functions related to object birthdays: $A(a)$ is the time at which a was first created, and $F(a)$ is the latest creation time of any object directly pointed to by a.

For efficiency, A and F values are maintained per-card rather than per-object, as discussed above. The A-value of a card is set when the card is used for the first time and never altered until the card is completely empty. The F-value is computed by remembering its value when we are scanning the pointers on the card during a collection. To maintain the F-values, cards with a valid F-value are write-protected. When a write occurs on a physical page, we turn off the protection and remember that the cards on that page no longer have valid F-values. At the next collection these cards will be considered to point into the threatened cards and so will be rescanned, recomputing their F-values.

A collection begins with the identification of a *threatening boundary* (TB), which is simply an integer between 0 and currentTime. Objects on cards with A-values strictly less than TB are immune; objects on cards with F-values strictly less than TB are bystanders. Thus, TB=0 does a full trace-and-sweep, TB>0 does a generational collection back some distance into the past.

To reduce card pollution, Collector II removes a card from consideration for allocation when it is three-fourths full and its A-value is two generations old. If it later becomes half full, it is again available for use by the allocator. To reduce paging overhead, Collector II also uses a version of trace-queueing, described in Section VI.

Results from Collector II

Measurements of Collector II used the same setup and benchmarks as measurements of Collector I, with the exception that cards were reduced to 512 bytes. Measurements of Collector II were done on a Sun-4/260, a faster machine than the Sun 3, with a significantly different architecture. Therefore, no direct comparison should be made between the times given in the previous section and those in this section.

We compared collection times for two different collection policies:

> **Policy 1**: Trace-and-Sweep. TB always 0, collect when heap is three-fourths full.
>
> **Policy 2**: Generational. TB = currentTime-1, collect every 100k bytes allocated.

For Boyer, the generational collector did 31 collections, using a total time for all GC's of 43 seconds. It touched on the average only 57 physical pages per collection. By comparison, the trace-and-sweep collector did 11 collections totalling 75 seconds of GC time, and touched an average of 285 physical pages per collection. Generational wins on every count.

For the compiler, our generational collector was more typical of generational schemes: it used more cpu, but touched fewer pages. It did 24 collections, using a total GC time of 46 seconds, touching an average of 126 physical pages at each collection. The trace-and-sweep collector ran only 4 times, using a total time of 29 seconds, touching an average of 430 pages at each collection.

The above numbers indicate that this collector is on the right track. The generational version of Collector II touches far fewer pages than the full trace-and-sweep version. Particularly remarkable is the use of less cpu time for generational collection than for full trace-and-sweep, at least in the Boyer benchmark. This is not usually true of generational collectors, which tend to trade cpu-time for a smaller working set. Overall, Collector II's times are still much slower than the normal Ibuki system, and slower than Collector I, but we have done none of the usual system tuning one needs in a production quality collector and allocator. We believe our collection times can be improved by a factor of 2-10 by some straightforward performance tuning.

VI. Other Fancy Tricks

There are many other tricks to making our collector implementations effective. Below we discuss four which are of particular interest: parallelism, trace queueing, lifetime prediction and application hints.

Parallelism

To reduce the pauses that result from garbage collections over large sets of objects, we have built an experimental implementation of Collector II in which tracing of the heap is incremental. The basic technique is to make a virtual snapshot of the heap when the collector starts. We do this by simulating a kind of "copy-on-write" behavior we write-protect the entire heap and copy pages as they are written by the application program during parallel collection. This is similar to techniques suggested by Shaw [Shaw87]. It takes only 3 milliseconds on a Sun-4 to take the write fault, copy the page, and unprotect it. In practice we have seen a maximum of only 20 page faults per second of application time (during the Ibuki compiler), although of course a much worse "toy" application is easy to construct.

Now when the collector is invoked it finds all of the pointers from the stack, registers, global data, and non-threatened cards into threatened cards. These pointers are remembered for tracing later. All cards are then write-protected, starting the virtual snapshot, and control is given back to the mutator. Objects allocated after the start of a parallel collection are not considered for collection.

Tracing now occurs incrementally during allocations. At each allocation request a single card of the heap is traced. Pointers off the card are saved for later. Notice that there is an upper bound on the time each allocation will take, and the expense of a large garbage collection can be amortized over a longer time.

Trace Queueing

It is a goal of generational collection to keep the size of the set of objects being traced small, thereby keeping collections short and unintrusive. Unfortunately, generational collection also allows accumulation of old but unreachable objects. In a long-lived program, such as an operating system, it is necessary occasionally to collect the entire heap. While this large collection is running, performance of the mutator degrades, but the program can continue running with no *a priori* limits on storage imposed by the collector.

If done naively, full collections can degrade mutator performance beyond usability. The problem arises in systems with virtual memory when the number of pages accessed by the collector is much larger than the number of available pages of physical memory. A naive depth-first trace of all live objects accesses nonresident pages frequently and repeatedly. Consequently, the collector generates a large number of page faults, swapping out the mutator's working set.

The trace of the heap must keep track of all objects that have been reached but not yet explored. A simple depth-first search keeps these unexplored objects on a stack and explores the object on the top of the stack when it finds itself at a leaf. Instead, the collector can partition the unexplored objects into buckets based on their addresses, and choose to explore objects that are already in memory whenever possible.

We implement trace queueing in Collector I by bucket-sorting the stack of references to be explored. In Collector II we trace all of the references on a single card at each allocation request. This gives the desired locality on the trace and a form of pseudo-parallelism. There is an upper bound on the time each allocation will take, since the collector is accessing only one card.

Trace queueing can cause unexpected performance penalties. For instance, the anomalous fact that Collector II uses less cpu time for generational collections than for full trace-and-sweep is largely explained by the greater number of off-page pointers followed during full trace-and-sweep. Turning off page queueing resulted in collection times more nearly proportional to the number of bytes actually traced by each collector.

Lifetime Prediction

With a copying collector, all new objects are allocated in a new space, and those surviving collection are copied to a different area. This results in a high density of live objects in memory. A non-copying collector, however, does not have this advantage. Objects that will soon die are allocated alongside objects that will be tenured. This natural intermingling of objects of different lifetimes results in fragmentation of the tenured storage. Allocating new objects in the holes in tenured storage fills the holes, but collecting those new objects is inefficient.

If we can predict object lifetimes in advance, then we can allocate long-lived and short-lived objects on separate cards. A reasonable predictor of object lifetimes is the allocation site. Static lifetime prediction algorithms typically rely on this assumption [Hudak86]. However, it is not clear exactly what constitutes an allocation site. The value of the program counter at the point of call to the memory allocation routine is a bad choice many LISP systems perform essentially all allocation from inside a routine implementing "cons." We need to identify the allocation site by information that simultaneously is cheap to compute and is a reasonable summary of the entire call stack. The stack pointer is a plausible candidate.

We built a modification of collector I that used the 11 low-order bits of the stack pointer to summarize the allocation site. It tracks lifetimes of a few objects, namely those that are the first allocation from a site and those that initiate allocation from a new card. If more than 3/4 of the tracked objects from a site survive the first collection, it declares the site to be long-lived. Cards that are more than half (but less than 3/4) full are reserved for allocations from such sites.

This scheme can be successful. On contrived test programs that exhibited exceptionally clean "generational" behavior, it eventually led to accumulation of a majority of all the long-lived objects on separate cards. This in turn led to significantly faster collection times, since the cards that were actually being examined by the collector contained few surviving objects. As a result, total collection times decreased to essentially those for the nongenerational collector.

Unfortunately, the success of the method is both machine- and application-dependent. It failed on a Sun 4, for example: on that processor, nearly every activation record has size exactly 96 bytes, so the stack pointer value contains little information.

On the compiler benchmark the strategy appeared to be marginally successful at concentrating long-lived objects, but not successful enough to show a performance improvement. Precise comparisons are difficult, since the partial collector effectively retires pages once they are 1/2 instead of 3/4 full. Nevertheless, at least the number of collections did not increase. On the Boyer benchmark some performance degradation was observed. However, given that this is a theorem proving benchmark, it is perhaps not surprising that object lifetime prediction is hard.

Application Hints

Our Collector II can accept hints from the application in the form of advice about TB values. For instance, the application can remember currentTime just before performing some storage-intensive operation. Then, when the operation is done, the application can request a collection back to the remembered time, which will reclaim the temporary objects created for that operation.

To test this idea, we made a run of Collector II compiling the two large lisp modules, with the following change: before compiling the first module currentTime was remembered, and between compiling the two modules a collection was done with TB of the remembered time. Total collection time was cut from 46 to 35 seconds still not as good as the trace-and-sweep (29 seconds) but much closer. Fascinating but not unexpected, the number of physical pages accessed during that collection in the middle was still only 129 pages, about the same as all the other generational collections. So with the right hints, at no loss of working set, collection time can be greatly improved.

VI. Conclusions

Our theory describes a large space of interesting garbage collectors. It is also a predictive theory so far, it has led us to two unique implementations of generational collectors with reasonable performance. There is much work to do to apply the theory to more kinds of collectors, and to explore further the space of implementations, particularily to take full advantage of the flexibility of partial orders.

VII. Acknowledgements

Hans Boehm was partially supported by DARPA/NSF grant CCR 87-20277. Barry Hayes was partially supported by the Northern California Chapter of ARCS Foundation, Inc., and by DARPA contract N000014-87-K-0828.

References

[Appel88] A. Appel, J. Ellis, and K. Li. Real-time Concurrent Garbage 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.

[Baker78] Henry G. Baker , Jr. List Processing in Real Time on a Serial Computer, *Communications of the ACM* 21, 4 (April 1978), pp. 280-294.

[Bartlett88] Joel F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots*. Western Research Laboratory Research Report 88/2, Digital Equipment Corp., February 1988.

[Boehm88] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. to appear in *Software: Practice and Experience*. 1988.

[Cardelli88] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow and Greg Nelson. *Modula-3 Report*. Olivetti Research Center Technical Report ORC-1, 1988.

[Courts87] Bob Courts. Improving Locality of Reference in a Garbage-Collecting Memory Management System. Internal TI memo, November 1987.

[Fitzgerald86] Robert Fitzgerald and Richard Rashid. The Integration of Virtual Memory Management and Interprocess Communication in Accent. *ACM Transactions on Computer Systems*. Vol. 4, No. 2. pp. 147-177, May 1986.

[Gabriel85] Richard Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, 1985.

[Hanson77] David R. Hanson. Storage Management for an Implementation of SNOBOL4. *Software: Practice and Experience*. Vol. 7, No. 2. pp. 179-192, March 1977.

[Hudak86] Paul Hudak. A Semantic Model of Reference Counting and its Abstraction. *Proceedings of the 1986 Conference on Lisp and Functional Programming*. pp. 351-363, Aug. 1986.

[IBUKI87] IBUKI Common Lisp, IBLC Release 01/01. IBUKI, Mountain View, Ca, 1987.

[Lieberman83] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, Vol. 26, No. 6, pp. 419-429, June 1983.

[Moon84] David A. Moon. Garbage Collection in a Large Lisp System. *ACM Symposium on Lisp and Functional Languages*, August 1984.

[Ripley78] G. David Ripley, Ralph E. Griswold, David R. Hanson. Performance of Storage Management in an Implementation of SNOBOL4. *IEEE Transactions on Software Engineering*, SE-4, No. 2, pp. 130-137, March 1978.

[Rovner85] Paul Rovner. On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language. Xerox PARC Report CSL-84-7, 1985.

[Shaw87] Robert A. Shaw. *Improving Garbage Collector Performance in Virtual Memory*. Computer Systems Laboratory Technical Report: CSL-TR-87-323, Stanford University, March 1987.

[Sobalvarro88] Patrick G. Sobalvarro. *A Lifetime-based Garbage Collector for LISP Systems on General-Purpose Computers*. Bachelors Thesis, Electrical Engineering and Computer Science, Massachusetts Institute of Technology. September 1988.

[Unger84] David Unger. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. in *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference*, 157-167, April 1984.

[Weiser89] Mark Weiser, Alan Demers, and Carl Hauser. The Portable Common Runtime Approach to Interoperability. *Proceedings 13th ACM Symposium on Operating System Principles*, December 1989.

[Wilson89] Paul R. Wilson. A Simple Bucket-Brigade Advancement Mechanism for Generation-Based Garbage Collection. *SIGPLAN Notices*, 24:5, May 1989.