Garbage Collection using a Dynamic Threatening Boundary

David A. Barrett and Benjamin G. Zorn

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430
{barrett,zorn}@cs.Colorado.EDU

Abstract

Generational techniques have been very successful in reducing the impact of garbage collection algorithms upon the performance of programs. However, all generational algorithms occasionally promote objects that later become garbage, resulting in an accumulation of garbage in older generations. Reclaiming this *tenured* garbage without resorting to collecting the entire heap is a difficult problem. In this paper, we describe a mechanism that extends existing generational collection algorithms by allowing them to reclaim tenured garbage more effectively. In particular, our *dynamic threatening boundary* mechanism divides memory into two spaces, one for short-lived, and another for long-lived objects. Unlike previous work, our collection mechanism can dynamically adjust the boundary between these two spaces either forward or backward in time, essentially allowing data to become untenured. We describe an implementation of the dynamic threatening boundary mechanism and quantify its associated costs. We also describe a policy for setting the threatening boundary and evaluate its performance relative to existing generational collection algorithms. Our results show that a policy that uses the dynamic threatening boundary mechanism is effective at reclaiming tenured garbage.

1 Introduction

Garbage collection is an effective programming language feature that has been used for many years in numerous languages. One successful enhancement to garbage collection algorithms is the use of generational garbage collection [14]. By exploiting the empirical property that the probability of a given object being garbage is higher for recently allocated objects than for older objects [2, 10, 19, 21], generational collectors improve reference locality and collector efficiency by avoiding tracing older objects during most scavenges. The success of generational collection algorithms is evinced by their commercial implementations in language environments that require automatic storage reclamation [1, 9].

SIGPLAN '95La Jolla, CA USA

Despite their success, generational collectors suffer from the *tenured garbage* problem. Specifically, all such algorithms occasionally promote objects that eventually become tenured garbage. Tenured garbage eventually needs to be reclaimed in some way, otherwise it will cause excessive memory consumption and an associated degradation in performance due to decreased reference locality.

The simplest solution to this problem is to occasionally collect all the older objects at a time when a long pause, possibly accompanied by many page faults, will not be too disruptive. Another solution is to set the garbage collector's promotion threshold (i.e., the age at which objects are promoted) to a fixed value that minimizes the tenured garbage. Unfortunately, both identifying acceptable times for long pauses, and identifying an appropriate promotion threshold, varies from one program run to the next and even during the lifetime of a single program. As a result, Ungar and Jackson propose a method called *feedback mediation* (FM) [17], which provides a partial solution to this problem by adjusting the promotion threshold for old objects based upon a pause-time constraint and object lifetime demographics. Their collector reduces tenured garbage by reducing the rate of object promotion. However, once objects are promoted using their method, the problem of reclaiming them still remains.

In this paper, we describe a mechanism that extends existing generational collection algorithms by allowing them to reclaim tenured garbage more effectively. In particular, our *Dynamic Threatening Boundary* (DTB) mechanism divides memory into two spaces, one for short-lived, and another for long-lived objects. Unlike previous work, our collection mechanism can dynamically adjust the boundary between these two spaces either forward or backward in time, essentially allowing data to become untenured.

To implement the DTB mechanism, all pointers that point forward-in-time must be recorded in the *remembered* set, unlike standard generational collectors, where only forward-in-time pointers that cross the generation boundaries are recorded. In this paper, we measure the overhead of maintaining the DTB remembered set and the cost of processing it during collection. Our measurements of allocation-intensive C programs indicate that the space overhead of the DTB ranges from 4–15% of the maximum storage required by the program, while the CPU overhead of maintaining the DTB ranges from 0–7% of total execution time (as measured by the number of instructions executed). We feel that the techniques proposed will be

most successful in two classes of languages: languages where pointers tend to account for a small fraction of the total memory allocated, and languages where most pointers do not point forward-in-time (i.e., where programs perform few destructive update operations, such as Standard ML). Our measurements in Section 5 indicate that C may belong in the first category.

Once the dynamic threatening boundary mechanism is available, it provides significant flexibility to the collector implementation and clearly separates issues of policy from implementation. In this paper, we investigate one policy, using the DTB mechanism, that attempts to reduce a program's tenured garbage. Our policy extends Ungar and Jackson's feedback mediation policy by taking advantage of the flexibility provided by the DTB. Our results show that this policy is more successful than feedback mediation at reducing tenured garbage and results in smaller program heaps when such tenured garbage exists, even when the additional space overhead of maintaining the DTB is taken into account.

2 Related Work

Generational algorithms [14, 15, 16] have proven successful at reducing the pause times and page fault rate of garbage collection [4, 6, 16]. Our work is based upon a formalization developed by Demers *et al* [6]. Their generational *Collector II* used a *threatening boundary* to divide memory into a *threatened* space for new objects, and an *immune* space for old objects, which were collected less frequently. To compare against non-generational algorithms, their collector modeled only classic generational collection by always setting the threatening boundary to the time of the previous collection. More recently, these authors have received a software patent covering their ideas [18]. Our algorithm expands upon theirs by using a new policy that dynamically adjusts the threatening boundary to limit resource consumption. Furthermore, we quantify the cost of a dynamic threatening boundary and also quantify the benefits in a collector that uses it.

One important policy for all generational collectors is when to *promote* objects from threatened space to immune space. Typically objects are promoted only after a fixed number of collections, specified as one of the tuning parameters made available to the application programmer. Ungar and Jackson [17] found that object lifetime distributions vary from one program to the next and often change as a program executes, showing that a fixed-age promotion policy will often be inappropriate. Instead, their feedback mediation collector promoted a number of objects only when a pause-time constraint was exceeded. Their simulations showed feedback mediation was successful at limiting pause times and indicated how memory usage increased as the pause-time constraint was reduced. This increased memory use, called tenured garbage, is caused by premature promotion of objects into the immune space when the collector must maintain a given pause-time. Unlike their algorithm, ours reduces tenured garbage by allowing objects to be demoted back into threatened space later when the collector pause-time falls.

Wilson and Moher's *Opportunistic Collector* [19] allocates objects created since the last collection in chronological order in memory. By selecting an appropriate address, only objects allocated since a specific time may be selected for promotion. However, once their collector has reclaimed objects from this new-object area, a different promotion policy must be followed because surviving objects are no longer in chronological order. Our algorithm preserves the object's allocation time for all objects, not just new ones, so ours may select ages among the surviving objects as well.

Like generational collection, our algorithm uses age as an indicator of when objects are most likely to die. When age is not a reliable indicator of garbage other methods must be used. Hudson and Moss [13] describe a *mature object space* that is collected incrementally based upon object connectivity rather than age. Likewise, Hayes [10] shows that when certain *key objects* die, they may indicate other unused ones as well. Like generational collectors, ours could eliminate objects from age-based collection by promoting them to mature or key object space, where they would be collected by other algorithms once they age enough.

The DTB mechanism we describe is an extreme case of a collection implementation that allows multiple generations (e.g., [5, 12]). As the number of generations grows to the number of live objects, the two concepts merge. From this perspective, our proposed collection policy (DTB_{dg}) represents a policy to select which generation to collect. This previous work has not quantified the overhead of maintaining large numbers of generations, either in terms of CPU cost or memory overhead.

3 The Dynamic Threatening Boundary Collector

In this section we describe the DTB mechanism and a policy for a collector that uses the it to reduce tenured garbage. We first describe how it provides the capability to dynamically adjust the boundary between old and new objects based upon the object allocation time. Next, we describe the new implementation issues raised by our mechanism. Then we discuss how the threatening boundary selection policy influences tenured garbage and pause times. Finally, we describe one policy that makes use of the DTB mechanism to trade available pause time for reduced tenured garbage.

3.1 The Dynamic Threatening Boundary Mechanism

Demers *et al* [6] have provided a useful formal framework for modeling generational garbage collection algorithms. As mentioned, their model partitions the object space into *threatened* and *immune* sets. Threatened objects are those that the collector traces to find unreachable objects and reclaim them. Immune objects are ones that will not be traced on this collection. The selection criteria for these sets distinguishes various collection algorithms.

Consider how a traditional generational collector selects its threatened and immune sets. The threatened set contains those objects that have survived fewer than a specified number of collections—typically one or two [1, 9, 19]. The root objects and all objects in older generations are immune. The *threatening boundary* divides the young threatened objects from the old immune objects. Each time the garbage collector is invoked, its policy sets the threatening boundary to the time of the kth previous collection, where k is a small integer constant. Scavenging the kth older generation corresponds to temporarily choosing a threatening boundary to the age corresponding to

the kth previous generation boundary. Generation boundaries simply constrain the set of allowable threatening boundaries.

Our mechanism eliminates generation boundaries. Instead, an explicit threatening boundary is established at the beginning of each collection. This boundary allows the collector to be much more flexible in choosing policies for selecting the threatened set.

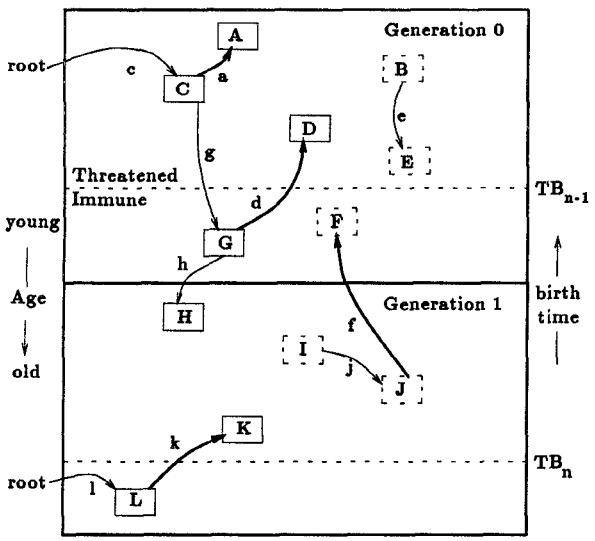

Figure 1: Dynamic Threatening Boundary vs Generations. The generational collector above divides memory into two generations, one young and one old. A dynamic threatening boundary collector adjusts a threatening boundary that may move between scavenges, say from TB_{n-1} to TB_n. Objects are shown ordered by birth time for exposition only; the actual implementation may maintain object locations in any order.

Figure 1 illustrates how the dynamic threatening boundary collector compares with other generational collectors. This figure shows a memory space divided into two generations. Age proceeds from youngest objects at the top of the page to the oldest at the bottom, whereas birth time increases in the other direction. Objects (in rectangles) are labeled in sequence by age. Arrows (labeled in lower case), indicate pointers between objects; heavy arrows indicate forward-in-time pointers.

For a generational collector, only pointer f must be recorded by the remembered set for Generation 0 because otherwise object F would be incorrectly deallocated by a scavenge of Generation 0. While the garbage objects B and E would be scavenged, objects I, J, and F would not; they are *tenured garbage*. Object F illustrates the phenomenon of *nepotism*: it remains alive even though it is threatened and unreachable because the tenured garbage points to it. Notice that once promoted, tenured garbage requires a complete scavenge of its generation to be reclaimed, in this case, Generation 1. A non-generational collector always collects all generations and so would collect all the garbage objects (B, E, F, I, and J) at the cost of tracing the entire memory space.

For a dynamic threatening boundary collector, a threatening boundary (shown by a dashed line at TB_{n-1}), divides the memory into *threatened* and *immune* spaces. Because the threatening boundary can be changed at the beginning of each scavenge, all forward-in-time pointers must be maintained in a single remembered set (pointers a, d, k, and f). At scavenge time only pointers that cross the threatening boundary are traced (pointer d). On a later scavenge, the collector is free to set the new threatening boundary to any desired time, say at TB_n. Unlike the generational collector, objects I, J and F become *untenured*, and will be reclaimed. Object K remains alive because pointer k references it from the remembered set.

3.2 Implementing the DTB Mechanism

In this section, we describe the implementation of the DTB mechanism. First, we state our assumptions; next we describe the implementation at a high level; and finally we discuss aspects of the implementation that are different from existing generational mechanisms.

For the purpose of this discussion, we assume that we are implementing the DTB mechanism in the context of a non-copying algorithm. This implies that objects are not relocated when collected and that the birth time of an object cannot be encoded in its address (as is the case in most copying algorithms). Thus, we assume that every object has a birth time field associated with it that is set when the object is allocated and never modified. We also assume that a data structure exists (the remembered set, which we describe in detail below), that indicates the location of every forward-in-time pointer. That is, every pointer stored in an object, o_1, that points to an object, o_2, such that the birth time of o_1 is less than the birth time of o_2 (o_2 is younger than o_1).

To understand how the DTB mechanism works, suppose that we are about to do a garbage collection and some policy has determined some specific threatening boundary. First, we augment the normal root set (i.e., registers and stack) with some of the elements in the remembered set. Specifically, we scan the remembered set and find all elements of it such that the forward-in-time pointer from the set points from an object born before the threatening boundary to an object born after the threatening boundary. The augmented root set now contains pointers to all objects in the threatened set that are reachable from the root set or immune set. We transitively traverse these roots marking all reachable objects in the threatened set. To complete the collection, we must then sweep the entire memory searching for unmarked objects that have a birth time later than the current threatening boundary. In practice, such a sweep can be deferred [20], reducing its performance impact.

The remembered set is maintained by the write barrier. In particular, when a pointer is written into an object, the birth time of the object being stored into and the birth time of the object being pointed to must be compared. If the store creates a forward-in-time pointer, then its location must be recorded in the remembered set.

The implementation of a dynamic threatening boundary mechanism relies mostly upon technology already available for other generational collectors. Here we describe new implementation issues raised by our mechanism and how they affect performance. These issues are maintaining object birth times, the effect of the remembered set on memory consumption, the

write barrier, and scavenging.

Object birth times must be available in order to determine the threatened set and to allow the write barrier to maintain the remembered set. The most straightforward implementation maintains a word per object. In environments where many small objects are allocated, objects may be co-located into pages (or areas) sharing the same birth time. Birth time may be chosen to be any appropriate metric of the granularity desired. One metric corresponding closely to existing generational collectors is the number of collections preceding the object allocation. A finer grained metric might be the total number of bytes allocated before the object was allocated.

Typically, a remembered set is maintained for each generation except the oldest; since our mechanism has only two generations, and the boundary between them moves, it uses a single remembered set instead. Generational collectors record only forward-in-time pointers that cross generation boundaries whereas ours records all forward-in-time pointers. Our remembered set is larger; we show how much larger in Section 5.

An inline write barrier may be used to filter each eligible store instruction and insert forward-in-time pointers into a hash table. Eligible store instructions are those that can be statically examined as possibly storing a pointer into the heap. For example, on the DEC Alpha under OSF1, we assume that stores to displacements from the stack pointer or to unaligned addresses are ineligible and do not require inline filtering. Eligible stores must compare the birth time of the source and target objects. If the target exceeds the source birth time, then the effective address of the store is inserted into the remembered set. Our algorithm performs this remembered set insertion more often than other generational collectors since we insert all forward-in-time pointers rather than just those that cross the threatening boundary. We show examples of how much more in Section 5.

Like all generational collectors, each element of the remembered set must be examined as an additional root during each scavenge. Our remembered set is larger, and we must perform an additional test to ensure we only trace objects that are born after the current threatening boundary. In Section 5 we show how much our mechanism would increase CPU compared to a traditional generational collector.

3.3 How Policies Affect Collector Performance

The choice of the threatening boundary affects both the CPU time spent scavenging and the memory wasted by tenured garbage. For a given collection interval, a young threatening boundary results in short trace times at the expense of more tenured garbage. An older threatening boundary wastes more CPU time tracing more of the live objects, but saves memory because older unreachable objects are reclaimed sooner.

Figure 2 shows how these values are related. The vertical axis is storage consumed (in bytes) and the horizontal axis is execution time (CPU instructions executed). Consider how a full garbage collection behaves. Periodically, at time t_i, a scavenge is triggered. The collector traces all the live storage and reclaims the rest. For example, at time t_1, Mem_1 bytes of storage were in use before the scavenge; the collector traced $Trace_1$ bytes, which included all the live bytes L_1. All the remaining bytes were reclaimed as shown by the curve dropping vertically to L_1

A generational collector scavenging at time t_{n-1} would only trace objects born after a fixed-age threatening boundary TB_{n-1}. This results in shorter pause times due to less storage traced, $Trace_{n-1}$, at the cost of more storage surviving, S_{n-1}. The difference between S_{n-1} and L_{n-1} is the tenured garbage.

At time t_n, the dynamic threatening boundary mechanism must select a threatening boundary TB_n before initiating scavenge n. The farther back in time TB_n is, the more storage will be traced, and the more garbage reclaimed.

3.4 A Policy to Reduce Tenured Garbage: DTB_{dg}

Here we describe additional assumptions made in our implementation and the specific details of our DTB_{dg} policy. We assume that garbage collection is triggered by a fixed amount of allocation (e.g., after 1 megabyte of allocation). Conceptually, the new data allocated since the last collection is considered to be in the "nursery," although in a non-copying collector such data will not be contiguous. We assume that the nursery is always collected in all the algorithms we consider. Thus, what differentiates the threatening boundary selection policies we consider is what, in addition to the nursery, is collected.

A non-generational collector always collects all data, which corresponds to selecting the threatening boundary to be 0 at every collection. We call such a collector FULL, and note that FULL never tenures any garbage.

A fixed-age generational collector promotes all objects that survive k collections, where k is a fixed value. If $k = 1$, (i.e., the FIXED1 policy used in our results) then this policy corresponds to tenuring objects as soon as they survive the nursery. If $k > 1$, then this choice corresponds to setting the threatening boundary a fixed distance backward in time.

Feedback mediation (or FM) is more sophisticated than a fixed-age policy. Instead of blindly setting the threatening boundary a fixed distance in the past, feedback mediation only advances the threatening boundary when a pause-time constraint (as defined by the number of bytes that are traced) is exceeded. In particular, if the current scavenge traces more bytes than a specified maximum, $Trace_{max}$, the threatening boundary is advanced, otherwise it is not and no objects are promoted. To advance the threatening boundary, the FM algorithm maintains a table of object demographics as it scavenges. The table classifies the currently scavenged objects by birth time into categories (or birth-time regions) and identifies how many bytes of objects are alive in each birth-time region. When the threatening boundary must be advanced, this table is scanned backwards starting at the current time. The scan accumulates the number of live bytes in each region until $Trace_{max}$ bytes is reached. This point determines where the new threatening boundary is set. In feedback mediation, demographics data are not preserved from one scavenge to the next, and information about objects born before the current threatening boundary are neither available nor used.

We seek a policy that reduces tenured garbage more effectively than feedback mediation. Our policy, DTB_{dg} (DTB demographic) attempts to do this by using the ability of the mechanism to dynamically adjust the threatening boundary to any desired value. The policy attempts to meet two goals: control pause times and minimize tenured garbage. Like feedback mediation, the policy attempts to meet the first goal by explicitly adjusting the threatening boundary forward to reduce pause times when it traces more than $Trace_{max}$ bytes. However, in

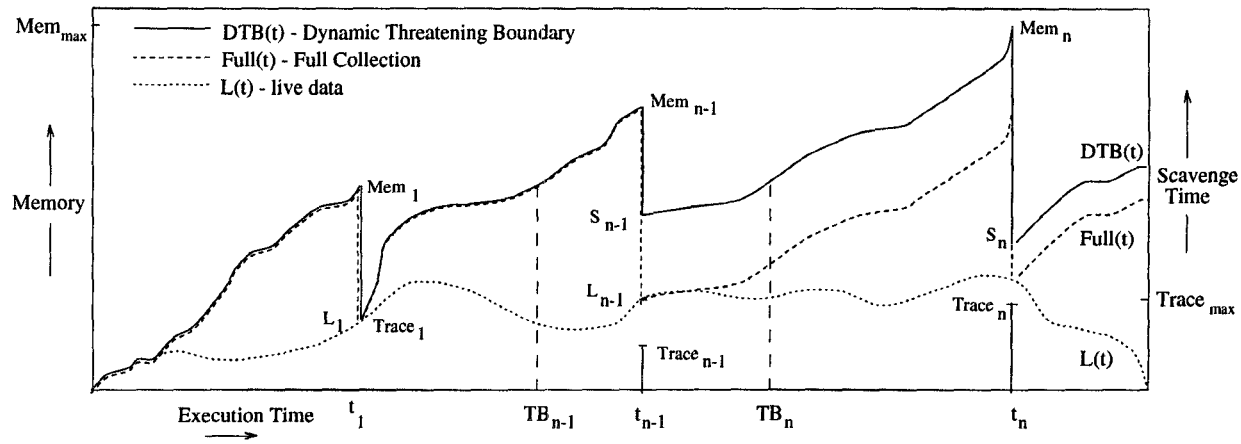

Figure 2: Garbage Collector Memory Use. A non-generational full garbage collector collects all garbage at periodic intervals as shown by curve $Full$ falling to curve L at time t_n. Like any generational collector, the dynamic threatening boundary mechanism saves tracing time by following curve DTB leaving some tenured garbage above the $Full$ curve. Notice that DTB reduced tenured garbage after time t_n by selecting TB_n to trace older objects than TB_{n-1} did at time t_{n-1}.

order to meet the second goal more effectively than feedback mediation it moves the boundary backward in time when it traces fewer than $Trace_{max}$ bytes. To do this adjustment, it preserves demographic information from one collection to the next and uses it in the same manner as feedback mediation, but without the constraint against moving the boundary backward.

Our policy has a couple of additional refinements. First, it attempts to predict how much data in the nursery will survive by estimating that the nursery survival rate will not change at the next collection. Therefore, it increments the sum obtained from the backward demographic scan by the number of bytes surviving from the nursery during the last scavenge. Second, it attempts to avoid repetitive tracing of long-lived clumps of old objects (the so-called *pig-in-the-python*). When it exceeds $Trace_{max}$ as a result of moving the threatening boundary backward to a given value, it does not select that (or an older) boundary again at the next scavenge. Each unsuccessful attempt to collect the clump doubles the intervening time until the next attempt on that clump.

Figure 3 illustrates how the threatening boundary, TB_n, is selected at time t_n. Each of the birth-time regions $A - D$ contains objects still alive at time t_n that were born between collections at times t_i and t_{i-1} (for integer $i = 1$ to n). Because region E has never been collected, the first refinement uses D's value as a prediction. The new treatening boundary, TB_n, is set to the oldest value that does not exceed $Trace_{max} = 100$ kilobytes (e.g., $E + D + C = 30 + 30 + 25 = 85 < 100$). TB_{n-x} illustrates a previous threatening boundary that was selected x collections ago as a result of the second refinement: collection of the long-lived region, A, was previously attempted, but resulted in too much storage being traced.

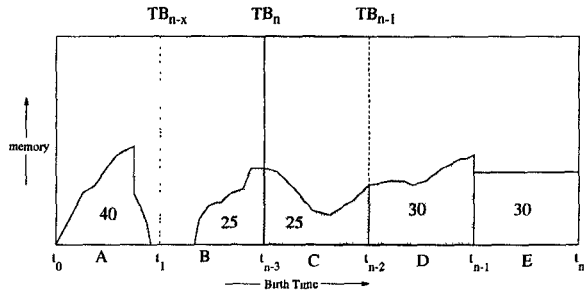

Figure 3: Threatening Boundary Demographics. The numbers underneath each curve show the total storage traced by previous collections for each of the birth-time regions $A - D$. Each region contains objects born between collections (t_{i-1} to t_i, $i = 1$ to n) that are live at time t_n. Region E, to be traced, uses the previous collection's value, D, as an estimate. At time t_n, the DTB_{dg} collector moves threatening boundary backward from TB_{n-1} to TB_n by using the oldest boundary such that the total traced area is less than $Trace_{max} = 100$ kilobytes ($E + D + C = 30 + 30 + 25 = 85 < 100$).

4 Methods

We are interested in comparing the relative performance of different garbage collection algorithms in terms of CPU and memory overhead, tenured garbage, and bytes traced. In order to determine the effectiveness of the dynamic threatening boundary mechanism and the DTB_{dg} policy, we instrumented a set of five allocation-intensive C programs using ATOM [8]. The programs are described in detail in Table 1 in Appendix A. We used memory allocation and deallocation events in these programs to drive a simulation of the different garbage collection algorithms. The output from the simulation consisted of memory and CPU usage patterns that were then processed to produce performance data.

We simulated each of four garbage collection algorithms: full collection (FULL), fixed-age generational (FIXED1), feed-

back mediation (FM), and our collector policy (DTB_{dg}). Scavenges were triggered after every 1 million bytes of allocation and $Trace_{max}$ was set to 100 thousand bytes.

In order to determine the CPU costs of our DTB mechanism, we wrote a program for the DEC Alpha that instrumented each of the applications and recorded the dynamic store frequency for various types of stores. Specifically, we distinguished between store instructions that can create pointers in the heap (which must be checked), and those that cannot (e.g. unaligned or stack pointer register displacements).

The exact store check overhead incurred for any collector depends highly upon the architectures of the processor, operating system, and collector implementation. We modelled the cost of an inline write-barrier for a non-copying conservative collector as might be used in a C or C++ environment. We assumed that a value can be checked to see if it is a forward-in-time pointer in 20 instructions, and if it is, that it takes 20 more to be inserted into the remembered set. (Hosking used 21 instructions to implement the store check on the MIPS R2000 [11]). The actual cost may vary significantly from our 20 instruction overhead, depending on the data structure used to lookup the descriptor for an object given a pointer that points inside it. For example, in version 4.0 of the Boehm-Weiser conservative collector [3], the store check would require two such lookups that would typically require 13 instructions each. Because the total store check overhead is directly proportional to the estimated cost of a single check, it is a simple matter to scale our store check CPU overhead estimates accordingly. Note that in any case, all generational collectors will incur this overhead.

To model the CPU overhead, we measured total instructions, total stores, eligible stores, and forward-in-time stores. We assumed that each entry in the remembered set required 20 instructions to be processed by the garbage collector during each scavenge.

To determine the memory required by the remembered set for the DTB mechanism, we measured the maximum number of live bytes, pointers, and forward-in-time pointers, for each application. We assumed that each remembered set entry required two pointer slots of eight bytes each (as on the DEC Alpha), one for the address updated, and one for a hash link. Other tradeoffs may be made to improve memory or time performance as appropriate. For the purposes of comparison, we assumed other generational collectors had a remembered set size of zero, incurred no costs to update it, and that our collector's remembered set was at the application's maximum at each scavenge. To determine the overhead for maintaining the birth times, we measured the maximum number of live objects and multiplied by eight bytes per object. To take into account memory fragmentation, the simulator allocated and deallocated dummy objects in a separate process using the GNU malloc/free routines as the simulation proceeded. We chose GNU malloc because it is among the most space-efficient allocation packages for C [7]. The maximum heap size was then recorded for each application.

5 Results

In this section, we investigate the costs of implementing the dynamic threatening boundary mechanism and the performance of our DTB_{dg} policy.

5.1 The Cost of the Dynamic Threatening Boundary Mechanism

As we have mentioned, a dynamic threatening boundary mechanism is similar to that of a conventional generational collector, except that it must maintain a remembered set of *all* forward-in-time pointers. We investigate the overhead of creating and maintaining that set in this section.

In Figure 4, we show the overhead of the store check, updating the remembered set (RS Insert), and scanning that set (RS Scan) during each collection. The numbers in the figure are shown in Table 2 and derived from the measurements contained in Table 3 in the Appendix. The store check overhead for an inline write barrier (Store Check) will be the same for DTB as it is for any generational collector. Clearly this portion of the total overhead can benefit from further optimizations such as identifying initializing store instructions. Such optimizations would benefit a generational algorithm and our DTB algorithm equally. As the figure shows, the additional overhead associated with maintaining the write barrier (RS Insert) and scanning the larger remembered sets (RS Scan) programs ranges from 0% to 7% of total program execution time.

Figure 5 shows the space overhead of maintaining all forward-in-time pointers in the remembered set and storing fine grain birth times for each object. The numbers for this figure are in Table 4 and are derived from Table 5 in the Appendix. Both the birth field overhead and the remembered set overhead vary widely across the applications but range from 4% to 15% of total memory consumption. GAWK and CFRAC overheads are dominated by the birth-time field because they have mostly small objects and low pointer density. ESPRESSO and SIS have much higher pointer density, so remembered sets dominate their overhead.

Because we are assuming a non-copying collector, we conservatively assume that an 8-byte birth field is included with each object. In practice, only several hundred distinct birth-time values are probably necessary at any time, and so this information could be encoded in many fewer bits (e.g., by mapping object addresses to an array of bytes). Because FM also collects and uses object demographic information, it also requires such birth-time fields. In a copying collector, more efficient methods of encoding birth information are also possible. By co-locating objects of the same age on the same page (or sub-page), the 8-byte overhead would be incurred per page instead of per object. Such an approach could suffer memory loss from internal fragmentation, however.

5.2 Evaluation of DTB_{dg} Policy

In this section, we evaluate the effectiveness of the DTB_{dg} policy in reclaiming tenured garbage and compare it to three other collection algorithms. We first consider total memory use, then tenured garbage reclamation, its effect on pause times, and finally, the total amount of data traced. Note that we considered the fixed costs of maintaining the write barrier and scanning the remembered sets for the mechanism in the previous section; here we evaluate one specific policy.

5.2.1 Memory Benefits

We first look at the memory required by each of the different collectors in Figures 6 and 7. In Figure 6 we evaluate

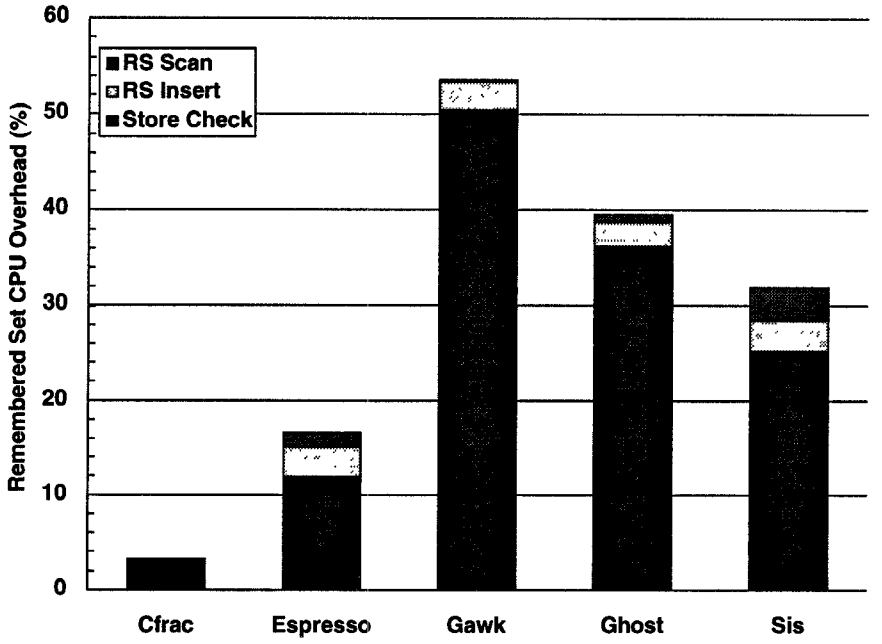

Figure 4: CPU Overhead of the DTB Mechanism. Maintaining the remembered set for generational collectors incurs CPU overhead from three sources: filtering at each store (Store Check), inserting forward-in-time pointers into the remembered set (RS Insert), and accessing the remembered set during each scavenge (RS Scan). The figure shows the additional overhead for each as a percentage of total instructions. The largest overhead is from the store check, which is identical for both DTB and any other generational collector that uses an inline write barrier. For comparison, we assume that a generational collector incurs no additional cost for scanning or updating the remembered set.

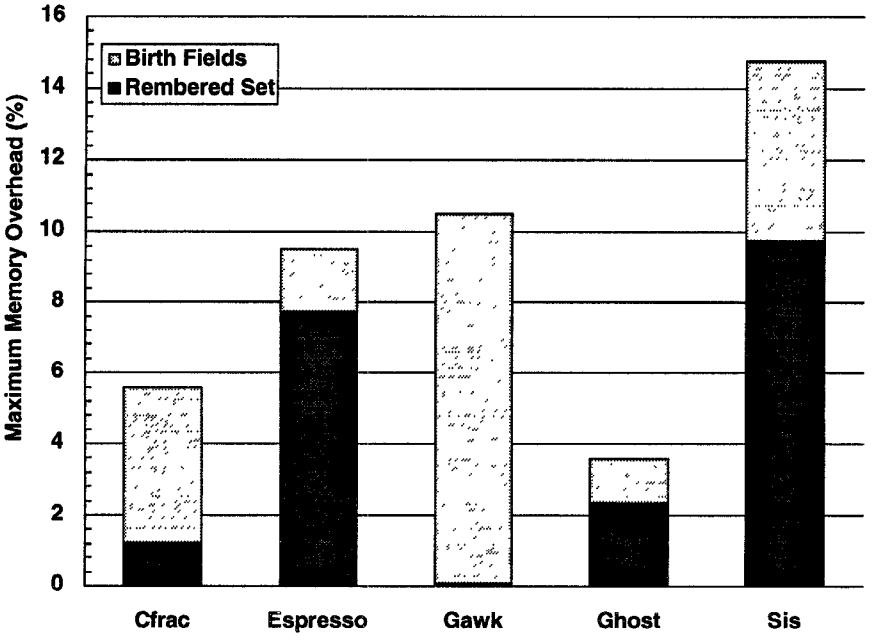

Figure 5: Memory Overhead of the DTB Mechanism. DTB incurs memory overhead for maintaining all forward-in-time pointers in the remembered set and a birth-time field for each object. The percentage of maximum memory consumed (including fragmentation) is shown.

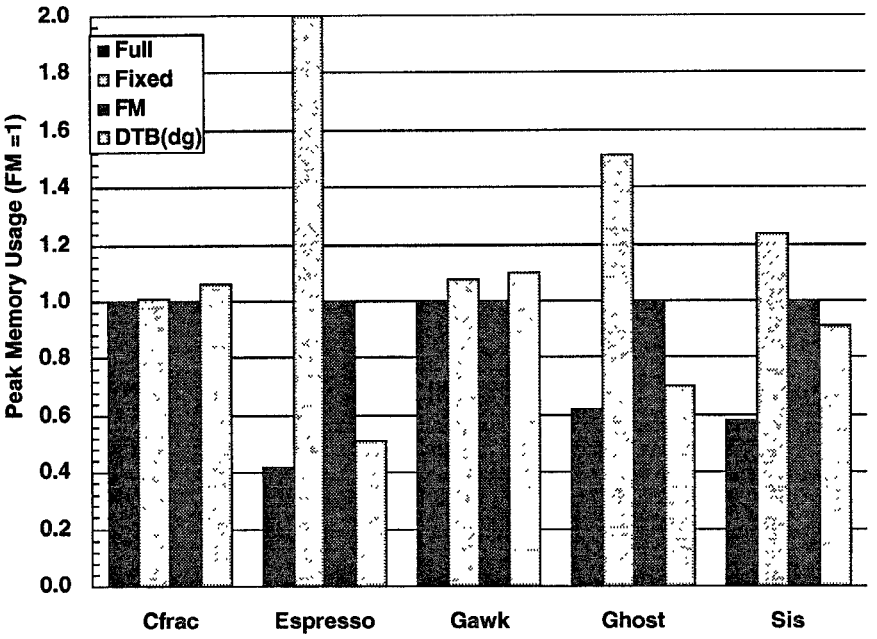

Figure 6: Maximum Memory Use (Relative to FM)
The maximum memory used by each collector for each application is shown. For easier comparison between applications, all values were normalized by dividing them by the maximum memory consumed by FM for each program. For DTB_{dg} the memory overhead shown in Figure 5 is included.

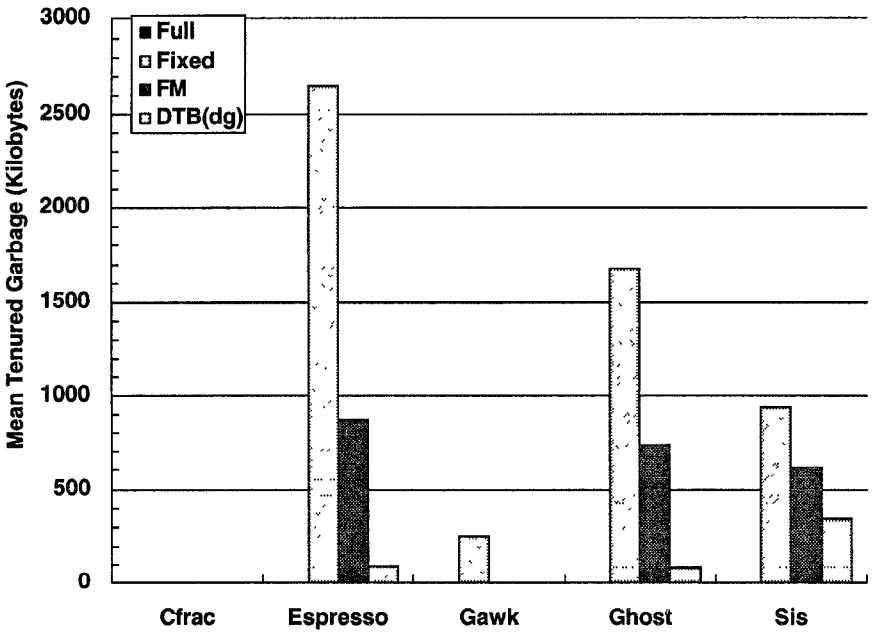

Figure 7: Mean Tenured Garbage
The figure shows the mean amount of garbage that was tenured by each of the different collection algorithms during program execution for the FM and DTB_{dg} collectors. FULL has no tenured garbage by definition.

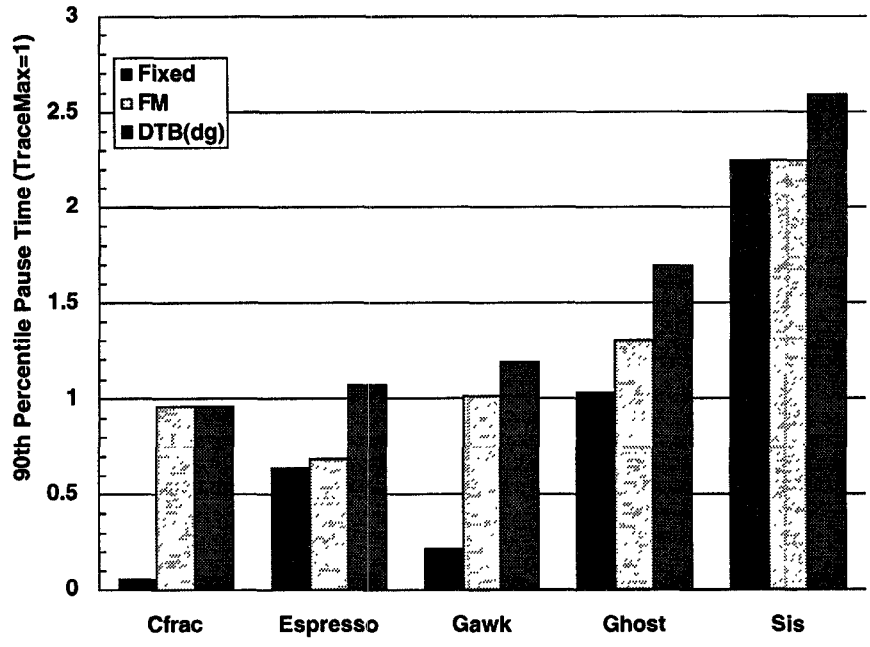

Figure 8: Pause Times
The figure shows the value corresponding to the 90th percentile pause time as measured by ratio of total bytes traced to a $Trace_{max}$ value of 100,000 bytes. FULL (not shown) has pause times off the chart (see Table 8 in the Appendix).

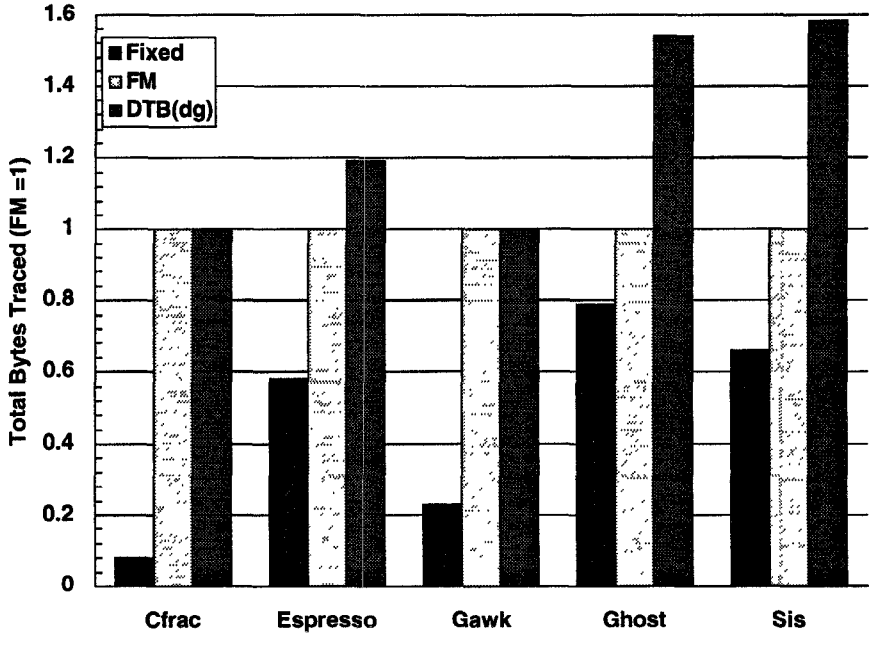

Figure 9: Total Bytes Traced
This figure shows the cumulative total of all bytes traced by each collector during the lifetime of each program. FULL is not shown because it was off the scale in all cases (see Table 9 in the Appendix).

collector performance with respect to maximum memory used (including fragmentation).

DTB$_{dg}$ saved significant amounts of memory for ESPRESSO and GHOST even after the effects of the remembered set and birth time overheads from Figure 5 have been included. SIS still had a modest improvement even though it had the largest memory overhead due to its high pointer density. Notice also that DTB$_{dg}$ incurs only a minor cost in CFRAC and GAWK where no collector distinguished itself.

Figure 7 shows how tenured garbage affected total memory use. CFRAC and GAWK do not generate significant amounts of tenured garbage (see also Table 7 in the appendix). Thus, there was no opportunity for either FM or DTB$_{dg}$ to recover much. For ESPRESSO and GHOST, however, tenured garbage was a significant portion of average memory use (50% and 30% respectively for FIXED1; 33% and 18% for FM). Observe how DTB$_{dg}$'s maximum memory use savings were roughly proportional to the proportion of tenured garbage. SIS had 22% tenured garbage for FM, but the savings were reduced by the 14% memory overhead of DTB (see Figure 5). When tenured garbage was present to a significant degree, our policy was able to collect much of it.

5.2.2 Object Tracing Costs

We now discuss the costs incurred as a result of collecting the tenured garbage.

Figure 8 shows the 90th percentile pause times in the test programs (as defined by bytes traced). FULL is not shown because it had pause times off the chart. The values for FM and DTB$_{dg}$ are much lower than FULL, showing the benefits of generational garbage collection. In SIS, the generational policies were not as effective at controlling pause times than they were in other programs, but they still saved 76% off the 90th percentile pause-time of FULL (not shown). From the figure, we also see what impact moving back the threatening boundary had on pause times. In particular, DTB$_{dg}$ was higher than FM and both were higher than FIXED1. As expected, the largest increase in pause times for DTB$_{dg}$ occurred for the same applications that had the greatest recovery of tenured garbage and were roughly proportional to the tenured garbage reclaimed.

5.2.3 Data Traced

Finally, we consider the amount of work done (e.g., time spent) tracing data in the different collectors. Garbage collectors trade reduced memory consumption for added object tracing overhead. Consider that FULL would be completely off the scale in Figure 9, and compare the FIXED1 collector in Figure 9 with the FULL and FIXED1 collectors in Figure 6. FULL always traces all objects, and thus has the lowest memory consumption and the traces the most bytes. On the other hand, FIXED1 tenures objects after just one collection, and thus has the lowest tracing overhead but uses the most memory of the two. If tracing overhead was the sole concern of users, the FIXED1 policy would be the obvious choice because it has the lowest overhead. Unfortunately, this algorithm has the property that tenured garbage accumulates (e.g., quite rapidly in ESPRESSO and GHOST) and its memory consumption is large, which motivates accepting the higher tracing overheads and pause-time overheads of FM and DTB$_{dg}$.

6 Summary

Generational garbage collection algorithms all occasionally promote objects that eventually become garbage. Reclaiming such tenured garbage is problematic, because in many implementations it requires collecting the entire heap.

We present the concept of a dynamic threatening boundary mechanism, which extends existing generational collection techniques. Unlike previous work, we describe a mechanism that can dynamically adjust the boundary between tenured and untenured data either forward or backward in time, essentially allowing data to become untenured at any time.

We describe an implementation of the dynamic threatening boundary mechanism and quantify its costs using measurements of allocation-intensive C programs. The central overhead of the dynamic threatening boundary is the CPU and memory costs required to maintain and scan a set indicating the location of all forward-in-time pointers. Our measurements show that maintaining this set adds 4–15% to the total space required by the program. Furthermore, we find that the CPU costs of maintaining and scanning this set add from 0–7% to the total execution time of the program over the costs of a conventional generational collection algorithm. We feel that implementing the dynamic threatening boundary mechanism is feasible in languages where programs often exhibit a low pointer density (e.g., as our measurements show, C) and in languages in which few forward-in-time pointers are created (e.g., Standard ML).

The dynamic threatening boundary mechanism is flexible and clearly separates issues of policy from implementation. We describe one policy, DTB$_{dg}$, for setting the threatening boundary based on an extension of Ungar and Jackson's feedback mediation collector. Because it attempts to reclaim tenured garbage, the DTB$_{dg}$ policy requires longer pause times and traces more bytes than feedback mediation. However, our results show the DTB$_{dg}$ policy, which uses a dynamic threatening boundary mechanism, is effective at reclaiming tenured garbage.

In the future, we plan to implement the DTB mechanism and carefully quantify its costs. We also plan to explore other threatening boundary selection policies in addition to the DTB$_{dg}$ policy evaluated here. Finally, we are interested in more carefully investigating the cost of implementing the write barrier in a conservative non-copying generational collector.

Acknowledgements

We would like to thank Hans Boehm, Urs Hölzle, Robert Rorschach, and David Ungar for making substantive contributions to the paper. We also thank the anonymous referees for their insightful comments. This work was funded in part by NSF grant No. CCR-9404669 and Digital Equipment Corp. External Research Grant Number 1580.

References

[1] Apple Computer Inc. *Macintosh Common Lisp Reference*, version 2 edition, 1992. pages 631–637.

[2] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 187–196, Albuquerque, New Mexico, June 1993.

[3] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, September 1988.

[4] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 157–164, June 1991. Toronto, Ontario, Canada.

[5] Patrick Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In Normam Meyrowitz, editor, *OOPSLA'86 Conference Proceedings*, pages 119–130, Portland, OR, September 1986. ACM.

[6] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 261–269, January 1990.

[7] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6):527–542, June 1994.

[8] Digital Equipment Corporation. *ATOM Reference Manual*, December 1993.

[9] Franz Inc. *Allegro CL User Guide, Version 4.1*, revision 2 edition, March 1992. Chapter 15: Garbage Collection.

[10] Barry Hayes. Using key object opportunism to collect old objects. In *ACM SIGPLAN 1991 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[11] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In *OOPSLA'93 Workshop on Garbage Collection in Object-Oriented Systems*, Washington, D.C., September 1993.

[12] R. L. Hudson, J. E. B. Moss, and A. Diwan. A language independent garbage collector toolkit. Technical Report COINS TR 91-47, University of Massachusetts, Amherst, MA, September 1991.

[13] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In *Proceedings of the International Workshop on Memory Management*, pages 388–403, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science vol. 637.

[14] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[15] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[16] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[17] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[18] Mark D. Weiser, Alan J. Demers, Daniel G. Bobrow, and Barry Hayes. Method and system for reclaiming unreferenced computer memory space. United States Patent No. 5,321,834, June 1994.

[19] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *ACM SIGPLAN 1989 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pages 23–35, New Orleans, Louisiana, October 1989.

[20] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *1990 ACM Conference on Lisp and Functional Programming*, pages 87–98, Nice, France, June 1990.

[21] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. *ACM SIGPLAN Notices*, 27(12):71–80, December 1992.

A Program Information

Program	Lines	Description
CFRAC	6000	Cfrac is a program that factors large integers using the continued fraction method. The input was a 28-digit number that was the product of two primes.
ESPRESSO	15500	Espresso, version 2.3, is a logic optimization program. The input used was the largest example provided with the release code.
GAWK	8500	GNU Awk, version 2.11, is a publicly available interpreter for the AWK report and extraction language. The input script formatted words in a dictionary.
GHOST	29500	GhostScript, version 2.6.1, is a publicly available interpreter for the PostScript page-description language. The input was a large PhD thesis. This execution of GhostScript did not run as interactive applications as it is often used, but instead were executed with the NODISPLAY option that simply forces the interpretation of the PostScript program without displaying the results.
SIS	172000	SIS, Release 1.1, is a tool for sythesis of synchronous and asynchronous circuits. It includes a number of capabilities such as state minimization and optimization. The input used in the run was one of the examples provided with the release (mcnc91/cse.blif). The operation performed was an attempt to reduce the depth of the circuit (`speed-up`).

Table 1: General information about the test programs.

	Store Check(%)	RS Insert(%)	RS Scan(%)	Total CPU(%)
CFRAC	3.17	0.00	0.04	3.21
ESPRESSO	11.97	3.07	1.48	16.53
GAWK	50.39	3.04	0.10	53.53
GHOST	36.23	2.54	0.70	39.48
SIS	25.23	3.20	3.46	31.90

Table 2: CPU Overhead of the DTB Mechanism. The table indicates sources of overhead in conventional generational collector and DTB mechanism based on an inline write barrier. All values are percentage of total execution time as measured by counting instructions. Store Check shows the overhead for checking for forward–in–time pointer stores; RS Insert shows the time to insert such pointers into the remembered set, and RS Scan shows the overhead during scavenging.

Program	Total Instr. (10^6)	Store Instr. (%)	Filter Instr. (%)	Forward Instr. (%)	Barrier Instr. (10^6)	CPU Overhead (%)
CFRAC	1471	6.55	0.16	0.000	47	3.2
ESPRESSO	2362	5.24	0.60	0.154	355	15.0
GAWK	1731	9.26	2.52	0.152	925	53.4
GHOST	1646	7.39	1.81	0.127	638	38.8
SIS	487	6.35	1.26	0.160	139	28.4

Table 3: Raw Data for CPU Overhead Computation. To determine CPU overhead, the data above was collected. All percentages given are percentages of total instructions executed. The Filter Instr. column shows the percentage of total instructions executed (Total Instr.) that require a filter to check for forward–in–time pointers. This percentage is small compared to the store instruction percentage (Store Instr.), which is itself a small percentage of total instructions. Forward Instr. shows the percentage of total instructions that required a forward–in-time pointer to be inserted into the remembered set. A model assuming 20 instructions for the filter, and 20 instructions for updating the remembered set predicts that a total of Barrier Instr. instructions will be required. The CPU overhead is computed from the ratio of Barrier Instr. to Total Instr. For ESPRESSO: $2362 \times (0.0060 \times 20 + .00154 \times 20) = 355, .15 = 356/2362$.

Program	Max. Heap Size (KBytes)	Max. Live Data (KBytes)	Rem. Set (%)	Birth Time (%)	Memory Overhead (%)
CFRAC	1536	123	1.21	4.39	5.60
ESPRESSO	1900	339	7.73	1.80	9.54
GAWK	8944	4720	0.06	10.43	10.49
GHOST	3684	1401	2.36	1.21	3.57
SIS	2928	672	9.76	4.99	14.76

Table 4: Memory Overhead of the DTB Mechanism. This table shows the additional memory required to maintain the remembered set and per–object birth times in the DTB mechanism. Max. Live Data is the maximum amount of live data used by the program at any time. Maximum Heap Size is Max. Live Data plus memory fragmentation overhead. Rem. Set indicates the ratio of space required for the remembered set to Maximum Heap Size for the DTB mechanism. Birth Time is the same ratio but for including a birth time field for each object. Memory Overhead is the total additional memory required for the DTB mechanism (Rem. Set % + Birth Time %).

Program	Total Allocated (KBytes)	Max. Heap Size (KBytes)	Max. Live Data (KBytes)	Max. Live Objects	Max. Live Pointers	Max. Rem. Set (Pointers)	Memory Overhead (%)	% Pointers in Heap
CFRAC	21183	1536	123	8624	3149	1189	5.60	20.04
ESPRESSO	181883	1900	339	4387	21451	9403	9.54	49.44
GAWK	235027	8944	4720	119403	1453	371	10.49	0.24
GHOST	101402	3684	1401	5697	35993	5562	3.57	20.07
SIS	45032	2928	672	18718	35884	18291	14.76	41.73

Table 5: Raw Data for Memory Overhead Computation. To determine memory overhead, the data above was collected. The memory overhead is modeled as maximum size of the remembered set (Max. Rem. Set) divided by the Max. Heap Size assuming two eight-byte pointers are required for each remembered set element. For ESPRESSO, $0.8 = 100 \times (9403 \times 16)/(1900 \times 1024)$. Max. Live Objects is used to compute the memory overhead for the per-object Birth Time field in Table 4 by assuming eight bytes per object. The pointer density (% Pointers in Heap) is the ratio of Maximum Live Pointers in the heap to Maximum Live Data expressed as a percentage assuming eight bytes per pointer. As the pointer density increases, so does the remembered set size and the corresponding overhead.

	FM (Kbytes)		FULL		FIXED1		FM		DTB_{dg}	
Program	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean
CFRAC	1536	1478	1.00	1.00	1.01	1.00	1.00	1.00	1.06	1.00
ESPRESSO	4108	2649	0.42	0.62	2.00	1.99	1.00	1.00	0.51	0.60
GAWK	8944	5237	1.00	1.00	1.08	1.06	1.00	1.00	1.10	1.00
GHOST	5484	4032	0.62	0.75	1.51	1.35	1.00	1.00	0.70	0.77
SIS	3712	2745	0.58	0.72	1.24	1.15	1.00	1.00	0.91	0.88

Table 6: Maximum and Mean Memory Consumed (Relative to FM). Each column shows the memory requirements of the different collectors measured. The first column shows absolute memory consumption of FM. Subsequent columns are normalized by dividing by this value. The DTB_{dg} column includes the memory overhead shown in Table 4.

Program	FULL (Kbytes)	FIXED1 (Kbytes)	FM (Kbytes)	DTB_{dg} (Kbytes)
CFRAC	0.00	3.04	0.04	0.04
ESPRESSO	0.00	2651.50	877.49	92.56
GAWK	0.00	253.03	0.03	0.03
GHOST	0.00	1680.11	734.07	84.11
SIS	0.00	941.76	618.45	344.16

Table 7: Mean Tenured Garbage. The table shows the average amount of garbage that was tenured by each of the different collection algorithms during the lifetime of the program. FULL cannot generate tenured garbage.

Program	$Trace_{max}$ (Kbytes)	FULL	FIXED1	FM	DTB_{dg}
CFRAC	100000	1.17	0.05	0.96	0.96
ESPRESSO	100000	2.19	0.64	0.69	1.07
GAWK	100000	43.62	0.21	1.01	1.19
GHOST	100000	13.36	1.03	1.30	1.69
SIS	100000	6.54	2.25	2.25	2.59

Table 8: 90th Percentile Pause Times (Relative to $Trace_{max}$). Each column shows the 90th percentile pause times for each program row. The first column shows the value of $Trace_{max}$. Subsequent columns are normalized by dividing by this value.

Program	FM KBYTES	FULL	FIXED1	FM	DTB
CFRAC	1529	1.29	0.08	1.00	1.00
ESPRESSO	6588	4.69	0.58	1.00	1.19
GAWK	21721	26.64	0.23	1.00	1.00
GHOST	5616	20.83	0.79	1.00	1.54
SIS	3767	6.93	0.66	1.00	1.58

Table 9: Total Data Traced (Relative to FM). The FM kilobytes column shows the number of kilobytes traced by the FM collector. Subsequent rows show the relative performance of the other collectors.