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

CU-CS-659-93 July 1993

University of Colorado at Boulder

Technical Report CU-CS-659-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

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
(303) 492-4398
{barrett,zorn}@cs.Colorado.EDU

July 1993

Abstract

Generational techniques have been very successful in reducing the impact of garbage collection algorithms upon the performance of programs. However, it is impossible for designers of collection algorithms to anticipate the memory allocation behavior of all applications in advance. Existing generational collectors rely upon the applications programmer to tune the behavior of the collector to achieve maximum performance for each application. Unfortunately, because the many tuning parameters require detailed knowledge of both the collection algorithm and the program allocation behavior in order to be used effectively, such tuning is difficult and error–prone. We propose a new garbage collection algorithm that uses just two easily–understood tuning parameters that directly reflect the maximum memory and pause time constraints familiar to application programmers and users.

Like generational collectors, ours divides memory into two spaces, one for short–lived, and another for long–lived objects. Unlike previous work, our collector dynamically adjusts the boundary between these two spaces in order to directly meet the resource constraints specified by the user. We describe two methods for adjusting this boundary, compare them with several existing algorithms, and show how effectively ours meets the specified constraints. Our pause–time collector saved memory by holding median pause times closer to the constraint than the other pause–time constrained algorithm and, when not over–constrained, our memory–constrained collector exhibited the lowest CPU overhead of the algorithms we measured yet was capable of maintaining a maximum memory constraint.

1 Introduction

As object–oriented languages such as C++ become more popular, more programmers are making heavier use of dynamic storage allocation. Garbage collection is a useful feature of programming languages because it allows the programmer to allocate storage dynamically without having to worry about reclaiming the storage once it is no longer being used. Although program development is easier with garbage collection, the resulting programs may have unacceptable performance when the memory usage patterns do not match

*This material is based upon work supported by the National Science Foundation under Grant No. CCR-9121269.

those anticipated by designer of the garbage collection algorithm used. The result is that programs may fail to complete because of memory exhaustion, excessive CPU overhead, or unacceptably long disruptive pauses while the garbage collector runs.

Generational garbage collection algorithms provide a partial solution to these performance problems. By making use of the observation that most dynamically allocated objects cease to be used very shortly after their creation [16, 8, 17, 3], generational collectors reduce pause times by reclaiming storage for recently allocated objects more often than older objects. The success of generational collection algorithms is evinced by their frequent use in language environments that require automatic storage reclamation [7, 1, 9, 4].

Despite their success, generational collectors must be tuned for applications that use memory differently than anticipated by the collector's designer. Typically tuning functions are provided that are expressed in terms of the operation of the garbage collector rather than the resource constraints and requirements of the application. For programmers not familiar with them, tuning generational garbage collectors is complex and time–consuming because determining how each of the tuning parameters affect resource consumption is difficult. To make matters worse, the application's programmer may not know the resource constraints under which the final program may be run by the user. For example, a compiler developer often knows neither the size of the program being compiled nor the memory available, nor even the relative importance to the user of speed versus memory consumption.

We propose a new garbage collection algorithm that is easily tuned to directly meet the resource constraints specified by the programmer or user. Like generational collectors, ours divides memory into two spaces, one for short–lived, and another for long–lived objects. Unlike previous work, our collector dynamically adjusts the boundary between these two spaces depending upon how well the specific resource constraints are being met.

For example, generational garbage collection trades increased space for reduced pause times by reclaiming only the portion of memory containing short–lived objects. The smaller this portion, the shorter the pause times, and the more memory is wasted by unused long–lived objects that are not reclaimed (*tenured garbage*). We would like to allow the user to select which is more important, reduced memory use, or shorter pause

times. Our algorithm does this by taking either a memory or pause–time constraint, and using it to select the size of the short–lived memory area based upon how well the constraint has been met so far during the current execution of the program.

This paper will describe our *Dynamic Threatening Boundary* algorithm, compare it to previous work, and present performance measurements. First, we discuss the previous work upon which this algorithm is based. Next, we introduce our model of garbage collection and use it to describe the details of our algorithm. Then, after discussing simulation methods, we present performance comparisons against other algorithms and show how well our algorithm met imposed resource constraints. Finally, we conclude with some observations about how these results may be used to improve future garbage collection technology.

2 Related Work

Generational algorithms [12, 14, 13] have proven successful at reducing the pause times and page fault rate of garbage collection [4, 6, 14]. 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. In order to compare with non–generational algorithms, their collector modeled only classic generational collection by always setting the threatening boundary to the time of the previous collection. Our algorithm expands upon theirs by using a new policy which dynamically adjusts the threatening boundary to limit resource consumption.

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 [15] 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 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 pause–time falls. Additionally, we allow a memory–constraint policy to be used instead if the user so desires.

Wilson and Moher's *Opportunistic Collector* [16] 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 [10] describe a *Mature Object Space* that is collected incrementally based upon object connectivity rather than age. Likewise, Hayes [8] showed that when certain *Key Objects* die, they may indicate other unused ones as well. Like generational collectors, ours could remove 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.

3 Background

A program initially has a number of live objects contained in an allocated set which grows as new objects are created as the program runs. Each object is created by allocating storage from the *heap*, and storing a pointer to it in one of the existing live objects. Eventually, all pointers to a set of objects may be overwritten and they become unreachable. When it is desirable to reclaim storage, a garbage collection is invoked and we say the collector *scavenges* the allocated set to find unreachable objects by *tracing* them and then reclaims the unreachable ones.

Tracing begins by identifying the *root set*, the set of all non–heap pointers into the heap. Root pointers may be stored in global variables, on the stack, or in registers; that is, in all objects directly reachable by the

program. Next, all heap objects pointed to by the root set are added to a set of reachable objects, either by marking them, or copying them to a reachable object space. Then, each new reachable object is examined for heap–pointers that are added to the original root set and the process repeats. Once all the reachable objects have been visited, the collector removes all the unreachable objects from the allocated set and reclaims their storage either by scanning storage for all unmarked objects, or by reclaiming all the storage at once in the case of a copying collector.

Generational collectors minimize the number of times each reachable object is traced during its lifetime by tracing old objects less frequently than young ones; once an object survives a few scavenges, it is likely to survive many more. Storage surviving several scavenges is promoted to the next older generation and only the youngest generation is scavenged at every collection. Successively older generations are scavenged less frequently because they grow more slowly and so longer–lived objects have more time to become unreachable. Tuning parameters select when to scavenge each generation and then number of scavenges an object must survive before being promoted.

In order to avoid having to trace objects in older generations for pointers into the scavenged generation, generational collection assumes that pointers from older objects to younger objects are rare. Such forward–in–time pointers into each generation are maintained explicitly in a collector data structure, the *remembered set*, that becomes an extension of the root set when that generation is scavenged. When a pointer store occurs to an object in a generation and points to an object in a younger one, the pointer location is added to the remembered set for the younger generation. Tracking such stores is called maintaining the *write barrier*. Stores from young objects to old ones are not explicitly tracked. Instead, whenever a given generation is collected, all younger generations are also collected.

Designers of generational collectors must also establish the appropriate size, and number of generations. The collector must determine how frequently to scavenge each generation; more frequent collections reduce memory requirements at the expense of increased CPU time because space is reclaimed sooner but live objects are traced more frequently. The space required by each generation is strongly influenced by the promotion

and scavenge policies. The effectiveness of all of these policies depends strongly upon the assumptions made by the designer about the allocation behavior of the programs using the collector.

The success of generational collection depends upon many aspects of program behavior. If older generation objects consume lots of storage, their lifetimes are long, they contain few pointers to young objects, pointer stores into them are rare, and many objects die at a far younger age, then generational collectors will be very effective. But, for some programs, which violate the policy decisions made by the collector implementor, performance may be unacceptable. Only the application programmer (or worse, the user) can identify these specific cases, and then he or she must learn about all the policy decisions described in the previous paragraph in order adjust each tuning parameter appropriately for their application. [1]

4 A Dynamic Threating Boundary Collector

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 [16, 1, 7]. 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 determined by a tuning parameter supplied by the application programmer. Scavenging the mth older generation corresponds to temporarily choosing a threatening boundary to the age corresponding

[1] One LISP manual [7] uses 27 pages to describe how to use various tuning parameters such as: auto-step, newspace, oldspace initial_oldspace, current-generation, tenure-limit, generation-spread, free-bytes-new-pages, free-percent-new, quantum, tenure, and tenured-bytes-limit.

to a the mth previous generation boundary. Generation boundaries simply constrain the set of allowable threatening boundaries.

Our algorithm 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. If the collector can arrange to be more effective, scavenging only objects that are most likely to be garbage, collection costs may be reduced.

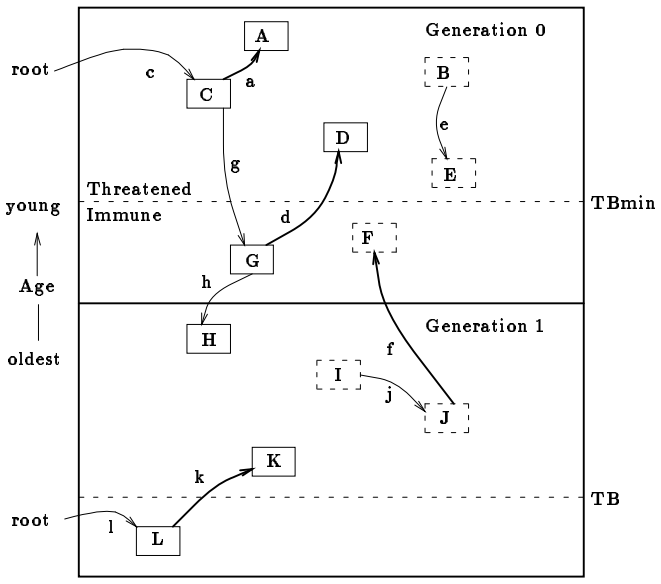

Figure 1: Dynamic Threatening Boundary vs Generations

The generational collector above divides memory into two generations, one young and one old. The dynamic threatening boundary collector adjusts a threatening boundary that may move between scavenges, say from TB_{min} to TB. Objects are shown ordered by age for exposition only; the actual implementation may maintain object locations in any order.

Figure 1 illustrates how the dynamic threatening boundary collector compares with generational collectors. This figure shows a memory space divided into two generations. Time proceeds from youngest objects at the top of the page to the oldest at the bottom; the objects (in rectangles) are labeled in sequence of their allocation time. 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 the dynamic threatening boundary collector, a threatening boundary (shown by a dashed line at TB_{min}), 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 d, k, and f). At scavenge time only pointers that cross the threatening boundary are traced (pointer d). Pointer a need never be recorded because the threatening boundary will never be placed younger than TB_{min} (it makes no sense for the collector to make almost all the objects immune). On a later scavenge, the collector is free to choose a different threatening boundary to any time desired, say at TB. 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.

As mentioned earlier, several policy decisions must be made by any generational collector: which generations to scavenge, the sizes of the generations and when objects are promoted from one generation to the next. The complex tuning parameters of generational collectors ultimately serve to answer just one question: what to collect. Once you establish what to collect, you must still decide when to collect.

These two issues are orthogonal, but since both cause time/space tradeoffs of a similar nature, they are easily confused. Increasing what is scavenged increases pause–times whereas scavenging more frequently reduces pause–times. Increasing either reduces garbage and increases CPU overhead. Wilson's Opportunistic Collector provides an answer for when to collect; our collector provides an answer for what to collect by mapping user constraints into a policy for selecting the appropriate threatening boundary.

4.1 How to Select the Threatening Boundary

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 live objects multiple times, but saves memory because older unreachable objects are reclaimed sooner.

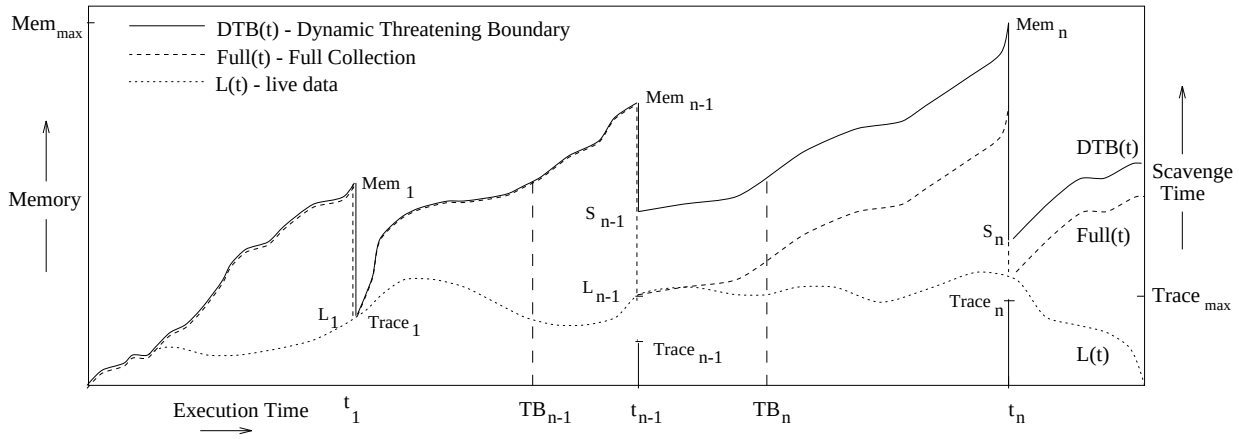

Figure 2: Garbage Collector Memory Use

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 collector 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}.

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 [2].

[2] For this discussion, we ignore memory fragmentation and time spent during garbage collection.

Collector	Threatening Boundary Policy
FULL	$TB_n \leftarrow 0$
FIXED1	$TB_n \leftarrow t_{n-1}$
FIXED4	$TB_n \leftarrow t_{n-4}$
FEEDMED	If $Trace_{n-1} > Trace_{max}$, $TB_n \leftarrow least(\{t_k \mid 0 \leq k < n$ and $t_k \geq TB_{n-1}$ and $Trace_{max} \geq \sum_{j=k}^{n-1} Born_j\})$ where: $Born_j$ = storage allocated after t_j but before t_{j+1}, and live at t_n else $TB_n \leftarrow TB_{n-1}$
DTBFM	If $Trace_{n-1} > Trace_{max}$, use FEEDMED else $TB_n \leftarrow t_n - (t_{n-1} - TB_{n-1}) \frac{Trace_{max}}{Trace_{n-1}}$
DTBMEM	$TB_n \leftarrow min(t_n \times \frac{Mem_{max} - L_{est}}{Mem_n}, t_{n-1})$, where: $L_{est} = \frac{1}{2}(S_{n-1} + Trace_{n-1})$ Mem_n = total storage used just before scavenge n S_{n-1} = surviving storage just after scavenge n-1

Table 1: Threatening Boundary Policy for Various Collectors

The Dynamic Threatening Boundary collector can model other collectors simply by altering the policy for setting the threatening boundary. Before the nth scavenge, at time t_n, the policy sets the appropriate threatening boundary, TB_n, which then traces $Trace_n$ amount of storage. DTBFM and DTBMEM correspond to our collector where $Trace_{max}$ is the maximum amount of storage to trace, or Mem_{max} is the maximum amount of memory to use. For a given implementation, pause–times are directly proportional to storage traced, so that a user–specified maximum pause–time is easily converted to $Trace_{max}$.

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 collector 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.

Depending upon which is more important to the user, we provide two policies for setting the threatening boundary, one for limiting maximum memory use to Mem_{max}, and another for limiting median pause times to $Trace_{max}$. Pause times are proportional to storage traced, so without loss of generality we represent pause times by the amount of storage traced. The following discussion describes the reasoning behind the formulas used for our collector policies as shown by the last two entries of Table 1.

Before each scavenge, at time t_n, our pause–time constrained collector, DtbFM, checks to see if the constraint was exceeded by the previous scavenge. If so, tracing is reduced to the desired value, $Trace_{max}$, by advancing the threatening boundary according to Ungar and Jackson's Feedback Mediation collector policy as shown by the FeedMed entry in Table 1. Otherwise, since it has an opportunity to reduce tenured garbage by tracing older objects, it will increase the number of bytes traced, $Trace_n$. It increases traced object age by lengthening the distance between the threatening boundary and the scavenge time by an amount proportional to the ratio of the desired storage traced, $Trace_{max}$ to the storage last traced, $Trace_{n-1}$ as shown by the DtbFM entry in Table 1.

Similarly, before each scavenge, our memory–constrained collector, DtbMem, sets the threatening boundary to achieve the maximum memory constraint, Mem_{max}, by controlling the desired amount of tenured garbage, Mem_{max} minus the live data, L_{n-1}. A conservative assumption is that the amount of garbage decreases linearly as the threatening boundary moves backward in time. The ratio of garbage to memory used (Mem_n) provides the suitable slope for this line. Unfortunately, without doing a full collection, the collector cannot determine precisely what L_{n-1} is, so it makes an estimate, L_{est}, by taking the average of the previous surviving storage, S_{n-1}, and the previous traced storage, $Trace_{n-1}$; it must lie somewhere between them. Since we always want to trace an object at least once, it sets the threatening boundary no later than the time of the previous scavenge, t_{n-1}. This policy is shown by the DtbMem entry in Table 1. Both collectors do a full collection on the first scavenge by setting the initial threatening boundary to 0.

4.2 Implementation Issues

The implementation of the dynamic threatening boundary collector relies upon technology already available for other generational collectors. Object birth times must be available in order to determine the threatened set and to allow a write–barrier to maintain the remembered set containing pointers to threatened objects.

Typically, a remembered set is maintained for each generation; since our collector 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. Like generational collectors, we assume that such pointers are a small fraction of all pointers, which ensures the remembered sets remain small. Our remembered set will be larger by an amount proportional to the ratio of forward–in–time pointers to inter–generational pointers. The sizes of remembered sets have not proven to be a problem for existing generational collectors.

Generational collectors use the generation containing each object to encode an approximation of its age. If you know the generation containing an object, and the promotion policy for moving an object to the next generation, you can derive the object's age from its generation. The precision with which you know an object's age is determined by the number of generations. If the allocation time is kept for each object, our collector can model a generational collector with an arbitrarily large number of generations. During a scavenge, only objects that are born after the threatening boundary are traced or reclaimed. If less precision is desired, (e.g., to maintain the write barrier using virtual memory) ages can be constrained arbitrarily and the same techniques used to implement multiple generations for other collectors apply to ours (e.g., Caudill's Smalltalk–80 implementation [5]).

5 Methods

In order to determine the effectiveness of the dynamic threatening boundary collector, we instrumented a set of four allocation–intensive C programs using Larus' trace generator *QPT* [11, 2]. The programs are described in detail in Tables 5 and 6 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 several garbage collection algorithms by setting the threatening boundary policy of our collector according to Table 1. We measured the CPU overhead, memory consumption, and pause times of the different collectors, assuming a machine that executes 10 million instructions per second, where the collector could trace 500 kilobytes per second. These simulation parameters were selected because they approximate those used by Ungar and Jackson to measure their Feedback Mediation collector [15]. Scavenges

were triggered after every 1 million bytes of allocation. The maximum pause–time was set to 100 milliseconds (50 thousand bytes traced) and the maximum memory constraint for DtbMem was 3000 kilobytes.

We are primarily interested in comparing the relative performance of the algorithms and measuring how well our algorithms tracked the pause–time or memory constraints. We assumed that the collectors had no memory fragmentation and that their CPU overhead was proportional to the number of bytes traced. Memory consumed for maintaining the remembered sets of the collectors was ignored for these measurements.

6 Results

In this section, we describe the results of simulating the six collection algorithms specified in Table 1 in each of the four test programs. In two of the programs, Ghost and Espresso, we present results from two different inputs. One goal of our evaluation is to compare the performance of the two dynamic threatening boundary algorithms we have proposed, DtbMem and DtbFM, with other existing algorithms. A second goal is to determine how well our algorithms met the programmer-specified maximum memory or pause-time constraints

We evaluate collector performance with respect to mean and maximum memory usage (assuming no fragmentation), median and 90th percentile pause times, and estimated CPU overhead due to tracing (see Tables 2, 3, 4). Table 2 also shows the mean and maximum memory usage of both the No GC algorithm (i.e., one that never invokes the collector) and Live, which reflects exactly how many live bytes exist during the program execution.

At first glance, it is clear that for the purpose of comparing the algorithms, Sis and Cfrac are less interesting than Ghost and Espresso. In particular, Sis has the behavior that much of what it allocates remains alive throughout its execution (i.e., compare the Live and No GC rows in Table 2). At the opposite extreme, Cfrac retains very little live data throughout its execution. In both cases, the program's behavior tends to reduce the differences in performance between the collectors.

Collector	Ghost (1) Mean / Max		Ghost (2) Mean / Max		Espresso (1) Mean / Max		Espresso (2) Mean / Max		Sis Mean / Max		Cfrac Mean / Max	
Full	1262	2065	1807	3033	564	1076	640	1188	4524	6980	497	992
Fixed1	1465	2453	2130	3632	667	1226	1577	2837	4691	7166	498	993
Fixed4	1262	2065	1807	3033	576	1088	760	1372	4524	6980	497	992
DtbMem	1460	2393	1984	3242	667	1226	1481	2365	4552	6980	498	993
FeedMed	1316	2125	1891	3168	620	1137	1095	1748	4691	7166	497	992
DtbFM	1265	2066	1839	3078	569	1111	695	1612	4691	7166	497	992
No GC	24601	49004	44243	87681	7874	14852	45428	104338	8346	14542	3853	7813
Live	777	1118	1323	2080	89	173	160	269	4197	6423	10	21

Table 2: Mean and Maximum Memory Allocated (Kilobytes)

Collector	Ghost (1) %ile		Ghost (2) %ile		Espresso (1) %ile		Espresso (2) %ile		Sis %ile		Cfrac %ile	
	50	90	50	90	50	90	50	90	50	90	50	90
Full	1743	2130	2720	4108	164	197	333	387	8165	11787	15	37
Fixed1	31	102	27	139	12	111	18	68	726	1609	5	7
Fixed4	120	334	150	409	20	192	28	137	2901	4545	15	22
DtbMem	34	112	200	1345	12	111	19	68	8165	11787	5	7
FeedMed	104	143	90	188	16	111	40	93	726	1609	15	37
DtbFM	106	168	97	234	53	178	93	364	726	1609	15	37

Table 3: Median and 90th Percentile Pause Times (Milliseconds)

Collector	Ghost (1) Traced / Overhead		Ghost (2) Traced / Overhead		Espresso (1) Traced / Overhead		Espresso (2) Traced / Overhead		Sis Traced / Overhead		Cfrac Traced / Overhead	
Full	40153	179.2	119011	203.7	1236	4.1	16389	14.0	57015	385.5	73	0.7
Fixed1	1373	6.1	2456	4.2	209	0.7	1615	1.4	6610	44.7	19	0.2
Fixed4	4610	20.6	8590	14.7	487	1.6	2878	2.5	24001	162.3	57	0.6
DtbMem	1489	6.6	23689	40.5	209	0.7	1662	1.4	50776	343.3	19	0.2
FeedMed	2641	11.8	4377	7.5	231	0.8	2642	2.3	6610	44.7	73	0.7
DtbFM	3026	13.5	5585	9.6	684	2.3	8201	7.0	6610	44.7	73	0.7

Table 4: Total Bytes Traced (Kilobytes) and Estimated CPU Overhead (%)

6.1 Meeting the Memory Constraint

Garbage collectors constantly trade memory usage for CPU overhead. Consider, for example, the FULL, FIXED1, and FIXED4 collectors in Tables 2 and 4. FULL always traces all objects, and thus has the lowest memory usage and the highest CPU overhead. FIXED1, on the other hand, tenures objects after just one collection, and thus has the lowest CPU overhead but uses the most memory. FIXED4 tenures after four collections, a more conservative tenuring policy, and thus falls between the FULL and FIXED1 on memory usage and CPU overhead.

If CPU overhead were 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 (fairly rapidly in GHOST, for example) and its memory usage becomes unbounded. The goal of the DTBMEM collector is to provide the CPU performance of the FIXED1 collector without letting the program memory usage grow without bound.

The DTBMEM collector attempts to match a maximum memory usage constraint supplied by the user. When the user supplies such a constraint, the collector is free to allow memory usage to grow until the constraint is met. In these programs, the collector was told to use a maximum of 3000 kilobytes of memory. Table 2 shows how well the DTBMEM met this constraint. In the two cases where it used more than 3000 kilobytes, GHOST (2) and SIS, the 3000 kilobyte limit was an over-constraint—that is, even the memory-optimal FULL algorithm was not able to operate with less than 3000 kilobytes. In both cases, the memory usage of the DTBMEM algorithm came within 7% of the FULL algorithm.

Table 4 shows that providing the maximum memory constraint allowed the DTBMEM algorithm to reduce its CPU overhead. In the cases where the 3000 kilobytes was not an over-constraint, the CPU overhead of the DTBMEM algorithm was quite similar to that of the fast FIXED1 algorithm. In the cases where 3000 kilobytes was an over-constraint, the CPU overhead of the DTBMEM algorithm increased as was necessary to try to meet the impossible constraint. In the case of SIS, we see that a much over-constrained DTBMEM algorithm degrades to the performance of the FULL algorithm.

6.2 Meeting the Pause-time Constraint

Users may want to limit the length of garbage collection pauses due to the nature of their application. Both the FeedMed and DtbFM algorithms allow users to specify a target pause-time and attempt to make collections take approximately that amount of time. In both cases, the algorithms react to pauses longer than the specified limit in the same way. Also in both cases, the best measure of whether the collector met the constraint is to look at the median pause time. Since the collectors are reactive to long pauses, a median that is close to what the user specified shows that the algorithm zeroed in on the specified value (i.e., half the collections took longer and half took less time).

The collectors differ when pauses take less than the user-specified amount of time. Where the FeedMed algorithm leaves the threatening boundary at the same place, the DtbFM algorithm attempts to move the boundary further back in time. As a result, the DtbFM algorithm should be better at making the median pause-time match the user-specified constraint. At the same time, the DtbFM collector should require less memory than the FeedMed collector because it will scavenge more older objects than the FeedMed collector.

Tables 3 and 2 show that the DtbFM collector was successful at achieving each of these goals. Table 3 shows that the median pause-time for the DtbFM collector was almost always closer to the 100 millisecond user-specified limit than the FeedMed collector. The Espresso application is an excellent illustration of the weakness of the FeedMed algorithm. In that program, the FeedMed pause-times were consistently less than 100 milliseconds, but because the algorithm was unable to push the threatening boundary back in time, it was unable to reclaim as much garbage as the DtbFM collector. As a result, the memory used by the FeedMed collector was often greater than that of the DtbFM collector, and sometimes significantly greater (e.g., in Espresso (2)).

Table 3 also shows the 90th percentile pause-times of the FeedMed and DtbFM collectors. In general, just as the medians are larger, the 90th percentile pause-times of the DtbFM algorithm are larger than

those of the FeedMed algorithm. However, the 90th percentiles are not so much larger in the DtbFM algorithm that the interactive response would be significantly worse than for the FeedMed algorithm.

7 Summary

Generational garbage collection is a powerful concept that has proven successful in a number of commercial language implementations. However, it is impossible for implementors of generational collection algorithms to anticipate the memory allocation behavior of all applications in advance. As a result, users are required to tune generational collection implementation parameters to meet the needs of their application. Unfortunately, in existing systems, correctly tuning algorithm parameters requires extensive knowledge of both the collection algorithm and the user program behavior.

In this paper, we present two variants of a garbage collection algorithm that each use a tuning parameter that directly reflects an easily–understood maximum memory or pause–time constraint. Like generational collectors, ours divides memory into two spaces, one for short–lived, and another for long–lived objects. Unlike previous work, our collector can arbitrarily select the boundary between these two spaces in order to directly meet the resource constraints specified by the user.

Based on the formal framework defined by Demers *et al* [6], we have shown how a dynamic threatening boundary collector can be used to meet a user–specified maximum memory or median pause–time constraint. Using trace–driven simulation we compared the two variants of the dynamic threatening boundary algorithm with existing algorithms, including Ungar and Jackson's Feedback Mediation [15]. We also show how the other algorithms we considered fit easily into the general dynamic threatening boundary framework.

Our results show that our memory–constrained threatening boundary algorithm meets the user–imposed memory constraint and uses available memory to reduce CPU overhead. We also show that the pause–time–constrained threatening boundary algorithm extends Feedback Mediation to exploit available pause–time and reduce memory overhead. In conclusion, our algorithms more accurately reflect user–imposed resources constraints and at the same time provide better performance than existing generational garbage collection algorithms.

A Program Information

GHOST	GhostScript, version 2.1, is a publicly-available interpreter for the PostScript page-description language. The inputs used were a large reference manual and a masters thesis. These executions 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.
ESPRESSO	Espresso, version 2.3, is a logic optimization program. The inputs used were examples provided with the release code.
SIS	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 (iscas89/s5378.blif). The operation performed was a verification with 1024 random input vectors.
CFRAC	Cfrac is a program that factors large integers using the continued fraction method. The input was a 25-digit number that was the product of two primes.

Table 5: General information about the test programs.

Program	Lines of Source	Execution Time (sec)	Total Allocation (megabytes)	Allocation Rate (kbtyes/sec)	Number of Collections
GHOST (1)	29500	31	49	1068	51
GHOST (2)	29500	71	88	733	90
ESPRESSO (1)	15500	62	15	240	16
ESPRESSO (2)	15500	240	104	435	107
SIS	172000	30	15	480	15
CFRAC	6000	8	3	402	4

Table 6: Allocation Behavior of Programs Measured.

References

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

[2] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[3] David Barrett and Benjamin 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.

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

[8] 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.

[9] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *ACM SIGPLAN 1992 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, pages 92–109, Vancouver, British Columbia, Canada, October 1992.

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

[11] Jamer R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report 1083, Computer Sciences Department, University of Wisconsin–Madison, 1210 West Dayton Street, Madison, WI 53706 USA, March 1992.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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