Leveled Garbage Collection

Guanshan Tong
IBM
2455 South Road
Poughkeepsie, NY 12601
Tel: (845)433-6277
email: gtong@us.ibm.com

Michael J. O'Donnell
Department of Computer Science
The University of Chicago
Chicago, IL 60637
Tel: (773) 702-1269
email: odonnell@cs.uchicago.edu

August 1999

Abstract

Generational garbage collection (GGC) is one of the most popular garbage collection techniques. GGC gains a performance advantage by performing *minor collections* on the younger objects in the heap, reducing the number of *major collections* of the whole heap. A *promotion policy* determines when an object moves from the younger generation to the older. The design of GGC has been justified by the plausible assumption that many objects die very young, and a few live a very long time. But, attempts to tune the performance of GGC by adjusting the promotion policy have been disappointing—only the simplest *immediate promotion* policy has proved attractive. The success of GGC is probably due to simplicity and to avoiding scans of the whole heap, rather than to accurate lifetime predictions.

This paper presents *Leveled Garbage Collection* (LGC), a new algorithm that is *not* based on object ages. It uses a heap structure and collection scheme similar to those of generational garbage collectors, and has a non-age-based promotion policy that doesn't promote all of the live objects, but still guarantees ample free space immediately after each garbage collection. By tuning LGC's promotion policy, we can often improve on GGC with immediate promotion.

Performance comparisons show that LGC outperforms GGC with immediate promotion policy in many cases, while losing only slightly on cases favorable to immediate promotion. LGC has a substantial advantage when the heap fits in main memory, and an even greater advantage as the heap gets paged to disk.

1 Introduction

Garbage collection is an important component of many modern programming languages. Since McCarthy's original work in 1960 [12], many different garbage collection algorithms have been developed [6, 2, 19]. Among the basic garbage collection techniques are mark-sweep [12, 20], mark-compact [7], and copying garbage collection [4, 8]. In the last decade or so, *generational garbage collection* (GGC) [11, 17, 13] has become one of the most important garbage collection techniques.

1.1 Generational Garbage Collection

GGC divides the heap into two or more areas, called *generations*, containing objects of different ages. For simplicity, we describe only the 2-generation case here. Objects are allocated in the younger generation, which is also the smaller. Each time the younger generation fills up, a *minor collection* reclaims garbage only in the younger generation, using a list of *back pointers* from the older to younger generation instead of sweeping the older generation. A *promotion policy* determines how many minor collections an object must survive before being promoted to the older generation. The simplest promotion policy is *immediate promotion*: all live objects are promoted at each minor collection. When the older generation fills up, a *major collection* collects the entire heap.

The efficiency of GGC depends on two key assumptions:

1. Most objects die before they are promoted; those that are promoted live a long time.

2. Most pointers go from younger objects to older objects.

When these assumptions hold, lots of storage is reclaimed during cheap minor collections, and more expensive major collections are much less frequent. When assumption 1 fails, GGC wastes time copying live objects from the younger to older generation, and collects the entire heap just as often as a nongenerational GC. When assumption 2 fails, minor collections become expensive, and may promote garbage due to back pointers from dead older objects. Assumption 1 is questionable [5, 3]. Assumption 2 may be violated by certain programs [19]. It appears that the success of GGC may derive more from its structuring of the heap and collections of different frequencies, than from the accuracy of its heuristic assumptions.

1.2 Leveled Garbage Collection

We propose a new method, *Leveled Garbage Collection* (LGC) [16], which ignores object ages, and aims instead at a good heuristic balance between freeing space in

the smaller top level and keeping objects there until they die. On our benchmarks, LGC has fewer major collections than GGC. We compare the wall-clock performance of LGC to GGC. The overhead of allocations is slightly higher for LGC than GGC, so GGC is slightly faster on some small problems. On large problems, LGC is much faster than GGC, particularly when the problems require a heap size that causes paging in major collections.

Rather than using assumptions about object ages, LGC tries to balance heuristically two conflicting desiderata for the smaller top level of the heap.

1. We want maximum free space in the top level for allocating objects.

2. We want to keep objects in the top level in hopes that they die and get reclaimed in a minor collection.

GGC with immediate promotion optimizes item 1: it frees all the space in the top level, at the possible cost of promoting objects that are about to die. GGC with multicycle promotion policies, however, may not recover enough free space after a collection. LGC guarantees that a specified amount of space is free after each minor collection. Subject to that constraint, LGC keeps live objects in the top level as long as possible.

During a minor collection, we do a tracing traversal [6, 19] to mark live objects. When the amount of live data encroaches the desired free space, we promote all further live objects to the bottom level. The precise identities of the promoted objects depend on the traversal order, not the object ages. The promotion of objects farthest down in the traversal is a plausible heuristic for minimizing the number of back pointers.

2 Overview of LGC

We describe 2-level LGC, but the method extends naturally to multiple levels.

2.1 Heap Organization

Let H be the total size of the heap. The heap is divided into two areas, the top level and the bottom level. All the allocations occur in the top level, and objects advance by moving from the top level to the bottom level. Each level is a single contiguous space.

2.2 Promotion Policy

There are two parameters, S which is the size of the top level, and F which is the amount of free space that must be available in the top level immediately after

each collection. Let $M = H - S$ be the size of the bottom level. As mentioned in Section 1.2, during the traversal, if the number of live nodes found so far exceeds $S - F$, we start to move live nodes found since then to the bottom level. If there are enough dead nodes, there is no promotion. The current LGC uses a depth-first traversal.

2.3 Tracking Interlevel Pointers

In order to garbage collect the top level without scanning the bottom level, we must keep track of all *back pointers* from the bottom level to the top level. We use a write barrier to detect the creation of back pointers, and use store lists [1, 19] to record the back pointers.

2.4 Minor and Major Collections

When an allocation request cannot be satisfied, a minor collection is invoked. A minor collection is a hybrid of mark-sweep and copying garbage collection. It starts out by traversing from the root set and the store list, marking and counting the live objects. When the counter reaches $S - F$, start to promote the remaining live objects by copying them into the bottom level. When the free space in the bottom level is not enough to accommodate the maximum amount of data that could possibly be promoted by the next minor collection (i.e., when $m + F > M$, where m is the size of allocated memory in the bottom level), a major collection is invoked. The major collection is a mark-compact process. It garbage collects the entire heap and compacts all the live objects into the bottom level.

2.5 Heap Expansion

After a major collection, if the ratio of heap size to the number of live nodes is lower than a desired value γ, the heap expands to the size of $\gamma * \#live$. Heap expansions are used to guarantee that there is enough free space in the bottom level for the promoted objects, and also to improve the performance when the heap is getting too crowded.

2.6 Performance Summary

LGC with $F = S$ is equivalent to GGC with immediate promotion. Multicycle GGC delays promotion by keeping nodes in the top level for a specified number of minor-collection cycles. Multicycle promotion policies complicate minor collection substantially [19], and it is unlikely that the retention of nodes until they die can pay for the loss of simple copying collection and the additional overhead to track

object ages. So, we compare LGC only with the immediate-promotion version of GGC, and show that adjustments to the parameter F in LGC can provide substantial improvements over immediate promotion.

An intuitive analysis of GGC vs. LGC shows that LGC makes at least as many minor collections as GGC with immediate promotion, but equal or fewer major collections. But, the overhead of allocation is higher for LGC. Our benchmarks are chosen, not to simulate a real load, but to show how the advantage of avoiding major collections balances the extra overhead of allocation. On problems where all objects are long lived, GGC outperforms LGC slightly. On problems where many nodes die soon after a minor collection, LGC wins by a substantial margin by avoiding major collections, even when the heap fits entirely in main memory. When the heap pages, the LGC win is magnified.

3 Some Design Details

3.1 Minor Collections

A minor collection is invoked whenever an allocation request cannot be satisfied from the remaining free memory. In the LGC system, the top level is used for allocation. We say that a minor collection is *effective* if it recovers enough free space for the pending allocation request. In a generational garbage collector with a multicycle advancement policy, a minor collection may fail to reclaim enough free space for the allocation, thus it cannot guarantee its own effectiveness. However, in the LGC system, a minor collection can guarantee its own effectiveness by making enough free space available in the allocation space immediately after this collection.

Naturally the collection plan is to start out by traversing from the root set and the store list, marking and counting the live objects. When the counter reaches $S - F$, start to promote live objects found since then by copying them to the bottom level. A depth-first tracing traversal is used, as it is required by the pointer reversal techniques used in the current LGC implementation. Future research may consider the use of a breadth-first traversal in the implementation. Live objects are marked in some way, either by setting a bit within the objects, or by recording them in a bitmap or some kind of table. These implementation details will be discussed in Section 4. The garbage reclamation phase is left implicit because spaces of unmarked objects are then the free space and can be used for allocation.

The description above implicitly states that the first $S - F$ live objects found are left where they were. One may wonder if it is worthwhile compacting them. On the one hand, compacting these live objects in the top level seems to be attractive. Without compaction, live objects and free space are interleaved, and thus the top level

has the fragmentation problem which has negative impact on allocations of objects of various sizes. Compacting can bring a contiguous free space, and thus eliminate the fragmentation problem and make the allocation easier. Compaction also brings better locality. However, this benefit will not be large for a small area like the top level, especially when taking into consideration that objects are often created in clusters that are typically active at the same time [10].

On the other hand, compacting these live objects in the top level has some serious problems:

- Compaction is expensive, and therefore it should not be done very often. However, the minor collection that collects the top level which is usually the smallest level on the heap will occur very frequently. Thus, compacting these live objects during each minor collection will be very costly.

- Since the live objects may be moved to a different location in the top level, back pointers stored in the objects of the bottom level must be updated. This requires scanning through the store list, finding all such back pointers and updating them. This is costly to perform, so we would like to avoid doing this after each minor collection.

Therefore, we choose not to compact the live objects that remain in the top level. In sum, the minor collection is basically a combination of mark-sweep and copying processes, where the sweeping phase is implicit and integrated with the allocation phase.

3.2 Major Collections

Generally, when the bottom level fills up, a major collection will be called for. However, the exact meaning of "fills up" merits further clarification.

For a minor collection, if a copying method with an immediate promotion policy is used, this minor collection may not be able to finish, because there may not be enough free space in the bottom level to accommodate the promoted objects. At this time, the minor collection has been half-done but is stalled now, and all the data structures are in an inconsistent state. It is very difficult if not impossible to do a garbage collection on the bottom level from this state, trying to get enough space recovered to let this minor collection proceed then.

Therefore, in general, each minor collection must be able to finish before another collection is needed. So, at the end of each minor collection, if the bottom level doesn't have enough free space to accommodate the maximum amount of data that could possibly be promoted by a single copying minor collection, a major collection should be called for. In the LGC system, during a minor collection, at most F live

space will be promoted to the bottom level. Thus, at the end of a minor collection, the bottom level will be checked to see if there is at least F free space still available, i.e., the condition $m+F \leq M$ still holds, where m is the amount of space already used in the bottom level and M is the size of the bottom level. If not, a major collection should be started.

We say that a major collection is *effective* if it frees at least F space. However, as will be discussed in Section 3.3.1, a major collection itself cannot guarantee its own effectiveness without heap expansion. Notice that a major collection should preserve the effectiveness of its preceding minor collection (Section 3.1), i.e., after this major collection, there will still be enough free space in the top level for the pending allocation request which invoked that minor collection.

Now, we discuss whether or not we should compact the bottom level after a major collection.

- Without compaction, live objects and free spaces are interleaved and thus there are fragmentation problems.

  When the minor collection promotes live objects, free space must be found in the bottom level to accommodate them. However, with fragmentation in the bottom level, finding space for objects of various sizes may be difficult and also expensive to perform.

- With compaction, fragmentation is eliminated and the bottom level has a single contiguous free space to accommodate objects promoted during the subsequent minor collections, for which finding space in the bottom level now is much easier. Therefore, the copying part of the minor collection can be simpler and faster. The minor collection is thus cheaper. Since it is frequent, the benefit is large.

- The improvement of locality by compaction may not be large for a small area such as the top level, but is important for the bottom level which is usually considerably larger than the top level and probably has to be paged.

Therefore, we would like to compact the bottom level. In fact, we should compact the entire heap, rather than just the bottom level:

- If only the live objects in the bottom level are compacted but the live objects of the top level stay where they were, all forward pointers from the top level objects to the bottom level objects need be found and then updated. This requires either maintaining an array of forward pointers or scanning the top level for such pointers at collection time. Either way is costly, and maintaining an array of forward pointers also takes space.

There will still be back pointers after the major collection, therefore the store list must be retained, updated during this bottom-level-only compaction, and cleaned up afterwards to kick out the invalid entries.

- Compacting the entire heap and moving all live objects into the bottom level allows us to discard the store list before a major collection, since it is not needed for a full-heap collection and it is empty after all objects are moved into the bottom level.

Therefore, for a major collection, we would like to compact the whole heap and move all the live objects into the bottom level.

3.3 Heap Expansion and Contraction Schemes

This section will investigate the value and effect of heap expansion and contraction, and will also discuss how to incorporate them into the design of the LGC systems.

3.3.1 Heap expansion

Each minor collection must be able to finish before another collection is needed. Therefore, at the end of each minor collection, if the bottom level doesn't have enough free space to accommodate the maximum amount of data that could possibly promoted by a single copying minor collection, a major collection should be called for. Let this maximum amount be denoted by A. In LGC systems, $A = F$, since during a minor collection at most F live data will be promoted to the bottom level.

However, will this major collection itself guarantee that there will be at least A free space available in the bottom level after the collection? The basic scheme of a major collection, which is to identify all the live objects (and compact them), doesn't provide such a guarantee.

Therefore, it is necessary to have a mechanism that not only can detect the situation that the bottom level doesn't have A free space available after a major collection, but can also establish the guarantee needed. Heap expansion schemes provide such a mechanism.

For effectiveness, we only need to expand the heap when the free space in the bottom level after a major collection is less than A. To reduce the number of major collections, it makes more sense to trigger expansion based on the proportion of live data, and to expand more than the minimum required for effectiveness. Let the ratio of the size of the bottom level to the amount of live data in the bottom level be γ. We control heap expansion with two parameters: γ_{min} and γ_{target}. If $\gamma < \gamma_{min}$ after a major collection, we expand the heap so that $\gamma = \gamma_{target}$. We choose γ_{min} so that, even with the smallest possible heap size, $\gamma \geq \gamma_{min}$ guarantees at least A nodes

free. There is a tradeoff in the value of γ_{target}. Too low a value leads to extra major collections. Too high a value leads to poor locality when nodes are scattered over a large heap.

As shown in Figure 1, we integrate heap expansion into the middle of a major collection. First, we scan for live nodes, then we determine whether and how much to expand, then we compact the live nodes into the expanded heap.

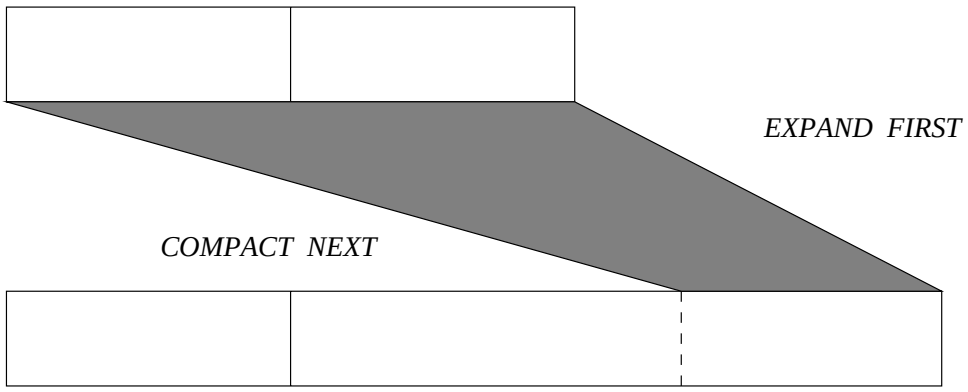

Figure 1: Compact after heap expands

In principle, heap expansion can adjust both the top and bottom levels. In our LGC system, we keep the top level size constant while expanding only the bottom level, because the top level should be sized to a high level of the memory hierarchy for better caching and paging performance.

3.3.2 Contract heap for better performance

Unlike the heap expansion which provides guarantee for the effectiveness of major collections, heap contraction is not required for correctness but it may improve performance. A γ greater than the desired value γ_{target} may indicate an overly large and underfull heap, on which live objects may scatter around and thus result in poor paging and cache locality. It would then be desirable to make the heap more compact, by reducing the heap size and keeping γ around γ_{target}.

However, it may not be a good idea to contract the heap unless the current γ is much larger than the desired value γ_{target}. Otherwise, the number of live data may rise soon after the heap is contracted, and a heap expansion may be needed then, so it may be better not to contract the heap at this point. Moreover, when a heap is contracted, it is probably a good idea to keep the resulting γ slightly higher than γ_{target}. This way, there is some room to accommodate the near-future increases in the number of live data so that a heap expansion won't be needed soon.

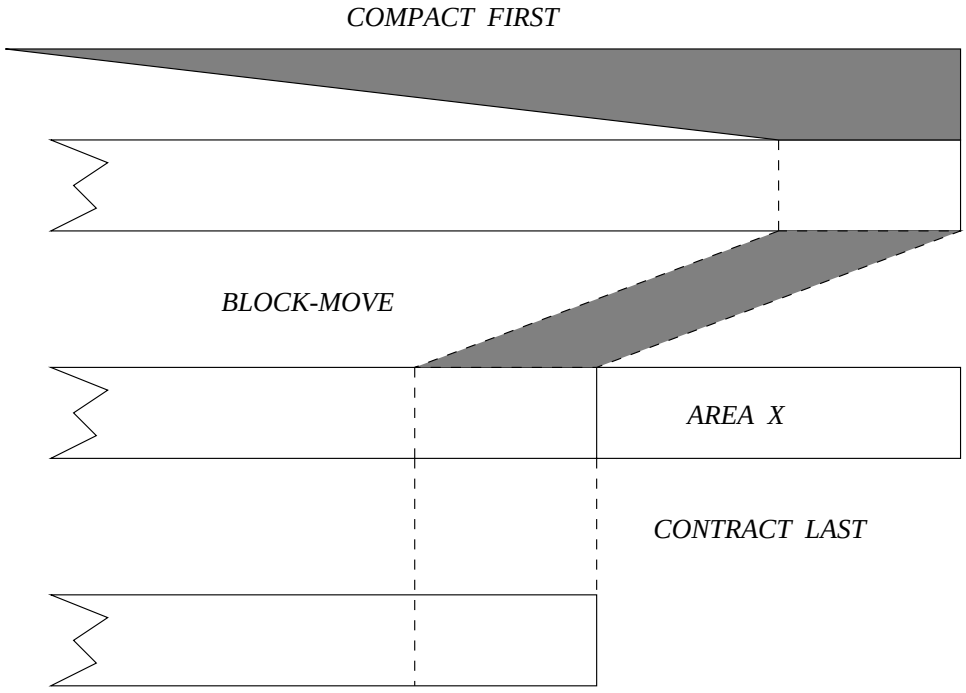

Figure 2: Compact before heap contracts

Heap contraction comes with costs, such as resizing each level, modifying the store list because of the boundary adjustments, and moving some objects because of the reduced heap size. These costs are not negligible, and thus in some cases a heap contraction may not be desirable. Furthermore, a smaller heap increases the number of garbage collections.

For a compacting major collection of LGC, unlike a heap expansion which should be done after the marking phase but before the compacting phase, a heap contraction, if desired, must happen after the compacting phase. Otherwise, the pointers into the released heap area (AREA X) would become dangling pointers and the subsequent compaction operation would produce unpredictable erroneous results. The appropriate sequence of operation is shown in Figure 2. Usually, when contracting the heap, we don't reduce the size of the top level, but contract the bottom level only.

3.4 Multilevel LGC Design Issues

The basic idea of two-level LGC extends to multiple levels. Number the levels $1, \ldots, n$ from top to bottom. In principle, we might collect any segment $[i, j]$ of contiguous levels, but we should probably limit ourselves to $[1, j]$ collections. A $[1, n]$ collection of the entire heap is a *global collection*. Further research is required to determine

good compaction policies, back-pointer representations, promotion policies, and expansion/contraction methods for multilevel LGC.

4 Implementation

We built a system for comparative studies of different garbage collection algorithms. This system is implemented in C. All objects are at aligned word (4-byte) addresses, and start with a one-word descriptor that has a 5-bit tag containing the object type and other garbage collection information and a 27-bit field specifying the length of the object. Benchmark programs are written in a Pascal-like language that is currently interpreted.

For LGC implementation, the pointer reversal techniques [18, 15] were used to implement both marking and compacting processes. By doing a little extra work during marking, compacting can be done in only one pass instead of two or more passes as in conventional methods, and without using any additional memory [16]. The mark bits are placed in the objects themselves.

Also, using a technique for maintaining the per-object mark bits [16], a mark-sweeping process doesn't need a mark-bit clearing phase before the marking, nor does it need an explicit sweeping phase which is instead done incrementally during allocation.

5 Performance

5.1 Method

To illustrate the value of LGC, in this paper, *we will compare LGC and a GGC on an otherwise identical base*: the heap size is the same and fixed, and so is the ratio between the top level (the younger generation in GGC) size and the bottom level (the older generation in GGC) size; the same implementation of minor and major collections which is described in Section 4 is used for both collectors. There are two differences between the implementations of the two methods:

1. LGC uses the non-age-based tunable promotion policy described in Section 1, while GGC uses the Immediate Promotion policy (GGCIP). This means LGC promotes only some of the live objects, while GGCIP simply promotes all of them.

2. We optimized the implementation of minor collection and allocation for GGCIP to take advantage of the simplicity of immediate promotion. Minor collection

becomes a pure copy process. Since the top level is empty after a minor collection, allocation merely advances a pointer through a contiguous free space.

The benchmarks in this paper are designed to show the tradeoff between the advantage derived by LGC from point 1 and the advantage derived by GGCIP from point 2.

In other work [16], we have compared the performance of LGC with that of Appel's elegant two-generation garbage collector [1], which is used in the current official release of the SML/NJ system. We found that LGC substantially outperformed this generational collector on a wide range of examples, and also had superior virtual memory performance. However, since both collectors can change the heap size dynamically, they end up working with different heap sizes, and the meaning of the comparison is less clear.

For both collectors, the top level size is one third of that of the bottom level. We evaluate their performance on several benchmarks for both the case where the heap fits into the available physical memory and hence no paging cost may be involved and the case where the heap is larger than the available physical memory and hence paging costs may be involved.

We use the following allocation-intensive programs as benchmarks:

1. **refresh**, which first builds a tree, then repeats a process that traverses the tree and replaces each node with a new node during traversal. This program is a variation of the benchmark "traverse" in the Gabriel suite [9], and is enhanced with intensive allocations.

2. **recruit**, which is similar to *refresh*, but during each traversal of the tree, only the tree leaves are replaced with new nodes. This program features a large number of long-lived objects.

3. **revcons**, which builds a list. In the course of list building, whenever a new item is appended to the list, a reversed copy of the resulting new list is produced, then the list building process continues from this reversed copy. When making the reversed copy, the original list is kept alive until the reversed copy is obtained.

4. **shrink**, which first builds a list, then builds a new list by copying every second item in the old list. Repeat this until the length of the new list becomes one. The whole process is repeated 10 times. This program is a variation of the benchmark "division by 2" in the Gabriel suite [9].

These benchmarks are used to test whether avoidance of major collections can balance extra allocation overhead.

Here, we explain some terminology. The *residency* of a program at a particular moment is the size of its live heap-allocated data [14]. The average residency was

obtained by forcing our leveled garbage collector to determine the amount of live data after each 0.5 MB allocation. We refer to the maximum residency of a program as its *problem size*. Each benchmark is in fact scalable in problem size. In our evaluation, a wide range of problem sizes is used for each benchmark.

Performance evaluation was done on a standalone Sun SPARCstation 4 (a 110 MHz microSPARC II CPU with a 16 KB virtual address cache) running Solaris 2.5.1, with 16 MB physical memory. The machine was physically disconnected from the network, and ran in single-user mode with no window system. Note that since the system also needs memory, the actual memory available to the benchmark programs is about 10-12 MB. We measured the total wall-clock elapsed time for each run. Time is the minimum of five runs. Note that the minimum and the maximum are very close together and their difference is usually less than 1%. Since most of the variation is probably in the operating system calls, the minimum is a good representative of the cost due to the GC method.

5.2 Results

We present the performance comparison results on the benchmarks *recruit* and *revcons* only. Results on the other two benchmarks are similar. The heap size is 10 MB for the case where no paging cost is involved, and 16 MB for the case where paging costs may be incurred.

For each figure, we show the best-performing F for LGC, whereas GGCIP (referred to as GGC below for brevity), by design, is an optimized version of LGC with $F = S$. More extensive benchmark results on different settings of F are reported in detail in [16]. A thorough analysis of the impact of the setting of F, and the best way to set default values, requires further research on a wider selection of benchmarks. From the benchmarks in [16], it appears that values between $0.3 \cdot S$ and $0.5 \cdot S$ are worth considering, and all of our benchmarks performed substantially better than GGC for large problem sizes with any F settings we evaluated between $0.3 \cdot S$ and $0.5 \cdot S$ [16].

5.2.1 No paging cost involved

Figure 3 shows the comparison result when running the benchmark *recruit*. In this figure, $F = 0.3 \cdot S$ for LGC, i.e., 30% of the top level must be free after each garbage collection. For both collectors, the top level is 2.5 MB and the bottom level is 7.5 MB.

On the smallest problem size 0.53 MB, GGC is slightly faster than LGC. Both collectors do not invoke any major collection. Since the top level size is 2.5 MB, LGC keeps all objects (including those nonleaf nodes that are long-lived) in the top level until they die and invokes 6 minor collections. With immediate promotion, GGC

promotes all the live objects to the bottom level during the first minor collection and then has more free space in the top level than LGC, so only 5 minor collections are invoked in GGC. LGC minor collection is cheaper than GGC's, since no copying (promoting) is needed. However, LGC has slightly higher allocation overhead, thus in total LGC is slightly slower.

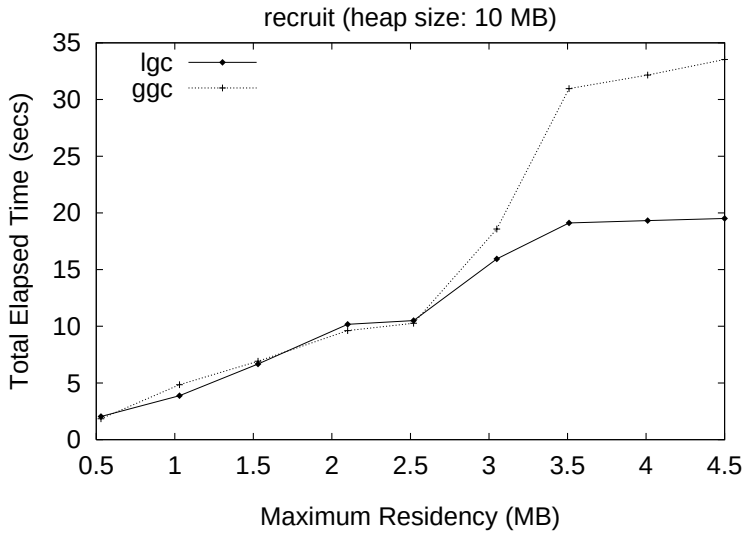

Figure 3: Promotion policy comparison: *recruit*

With a little bit bigger problem size, i.e., 1.03 MB, LGC is still able to keep every object in the top level, whereas GGC has to invoke a major collection since every live object is promoted in each minor collection and the bottom level is eventually filled up. A major collection costs substantially more than a minor collection, and this time LGC is 20% faster than GGC.

As shown in Figure 3, on medium to large problem sizes, LGC performs much better than GGC, particularly at large problem sizes. Although GGC immediate promotion policy can quickly move those long-lived nonleaf nodes out of the top level, the premature promotion problem becomes serious at large problem sizes. For example, when the problem size is 4.50 MB, there are 1.56 MB leaf nodes and 2.94 MB nonleaf nodes. When the nonleaf nodes are on the bottom level, since leaf nodes are repeatedly replaced, then at a minor collection time, the current leaf nodes are all in the top level. For GGC, all of them have to be promoted and this gives a promotion amount of 1.56 MB. Then, the whole tree is in the bottom level and now the free space in the bottom level is only $7.50 - 4.50 = 3.00MB$. This space can sustain only one more minor collection before a major collection must be invoked (because this

free space will be 1.44 MB then and not enough for another 1.56 MB promotion). This major collection then empties up the top level, but still leaves just 3.00 MB free space in the bottom level (there is 4.50 MB live data on the heap). So, the same situation continues and a major collection is needed after each minor collection. In total, GGC invokes 44 minor collections and 42 major collections.

In contrast, LGC can keep most of these short-lived leaf nodes in the top level until they die. The first major collection sends all nonleaf nodes to the bottom level. Then, at each minor collection, since the live data is only 1.56 MB, on a top level of 2.5 MB and with $F = 0.3 \cdot S$, there is nothing to promote. Therefore, LGC invokes only one major collection in total. Each minor collection recovers 0.94 MB instead of the whole top level as in GGC, so LGC has 122 minor collections. However, with nothing to promote (copy) and just marking live objects, a LGC minor collection is even cheaper than a GGC minor collection which needs to promote all these objects in addition to tracing them. So, LGC is much faster than GGC. Therefore, this is a battle of how to heuristically balance between freeing space in the smaller top level and keeping objects there until they die in hopes that they will be reclaimed in cheap minor collections. On this benchmark, LGC promotion strategy shows advantage over GGC immediate promotion policy.

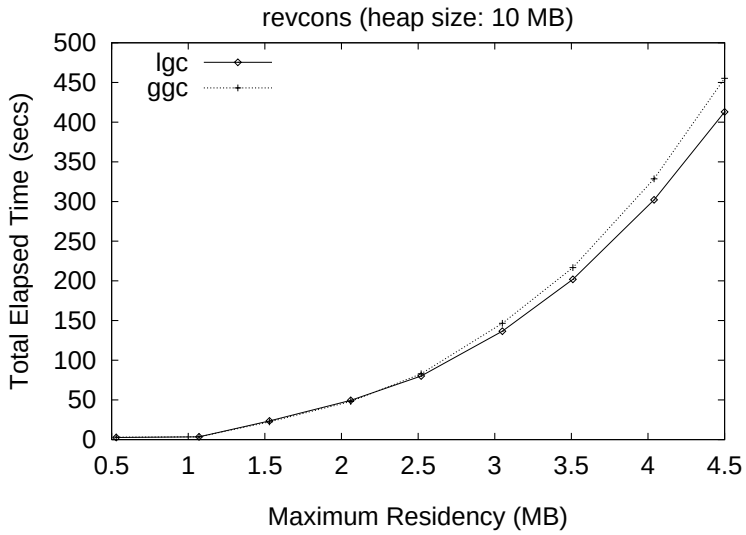

Figure 4: Promotion policy comparison: *revcons*

Figure 4 shows the comparison result when running the benchmark *revcons*. In this figure, $F = 0.3 \cdot S$ for LGC. The general trend in this figure is similar to that in Figure 3, though the performance difference between LGC and GGC on large

problem sizes is not as large. LGC is faster than GGC on the smallest problem size 0.53 MB, because LGC can keep all objects in the top level and save the promotion (copying) effort incurred in GGC. GGC promotes every live object at each minor collection and invokes one major collection. When the problem size is 1.53 MB, LGC promotion policy leads to too frequent minor collections (233 versus 156 in GGC), and the overall cost becomes slightly higher than that of GGC. However, on large problem sizes, LGC promotion policy starts to show its merits by outperforming GGC immediate promotion policy.

To summarize, our performance evaluation shows that when no paging cost is incurred, LGC promotion policy has an advantage over GGC immediate promotion policy, particularly for large problem sizes.

5.2.2 Interaction with virtual memory system

Now, we study how LGC and GGC promotion policies interact with the virtual memory system. Notice that on a heap of 16 MB, a major collection will incur paging costs. For both collectors, the top level is 4 MB and the bottom level is 12 MB.

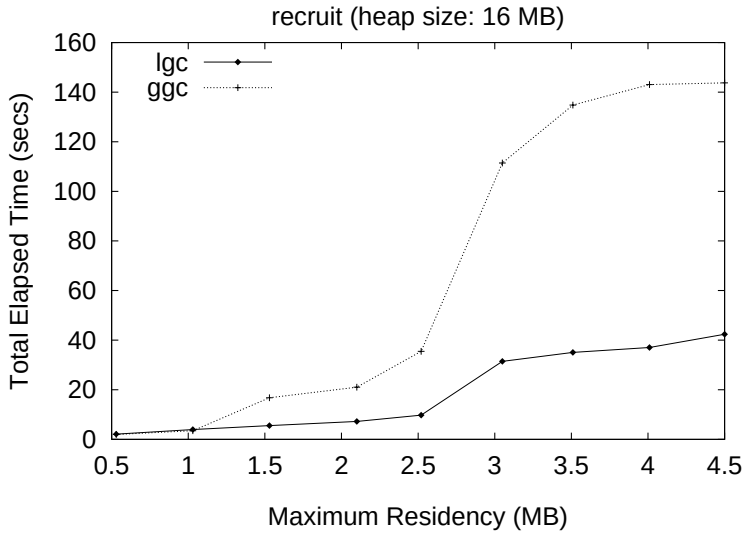

Figure 5: Paging performance of promotion policies: *recruit*

Figure 5 shows the result on the benchmark *recruit*. For LGC, $F = 0.5 \cdot S$. On the two smallest problem sizes 0.53 MB and 1.03 MB, LGC is slower than GGC, but only very slightly. On a heap of 16 MB, with such small problem sizes, both collectors

just invoke a few minor collections and need no major collection. However, LGC has slightly higher allocation overhead. Thus LGC overall cost is a bit higher.

On all other problem sizes, LGC substantially outperforms GGC. In particular, on larger problem sizes, LGC is more than three times faster than GGC. Paging costs incurred during major collections account for this performance difference. For *recruit*, after the nonleaf nodes are promoted to the bottom level, the working set of the program execution will consist of the pages that contain the top level and the pages for the bottom-level area that contains those nonleaf nodes. If the size of this working set is larger than the available memory, extra paging costs will be incurred during the program execution. However, in the current setting, since the top level size is 4 MB, even for the largest problem size 4.50 MB which has about 3 MB nonleaf nodes, the size of the working set is only 7 MB, which will not incur any paging cost during the program execution. Therefore, all paging costs incurred in Figure 5 are due to major collections.

For example, when the problem size is 1.53 MB, GGC invokes one major collection while LGC does not. Even though LGC has 20 minor collections while GGC has only 13, LGC is still 3 times faster than GGC. This shows that the paging costs caused by one major collection is overwhelming compared with other garbage collection costs.

When the problem size is 3.05 MB, one major collection is invoked in LGC and hence there is a noticeable increase in LGC total cost. For GGC, as explained before, after the nonleaf nodes are promoted to the bottom level, during each minor collection the current leaf nodes will all be in the top level and have to be promoted. However, they die soon after being promoted (as they are replaced by new leaf nodes generated in the top level). This premature promotion forces many major collections. For example, GGC has 6 major collections on the problem size 3.05 MB. In contrast, after promoting the nonleaf nodes to the bottom level, LGC is able to keep all of the short-lived leaf nodes in the top level until they die. Therefore, even on the problem size of 4.50 MB, LGC needs only one major collection in total.

Figure 6 shows comparison result when running the benchmark *revcons*. For LGC, $F = 0.3 \cdot S$. *revcons* allocates heavily on large problem sizes, and this forces a large number of minor collections and also many major collections in both collectors. However, LGC promotion policy leads to *much* fewer major collections than GGC immediate promotion policy. Therefore, LGC incurs much less paging costs than GGC, and hence outperforms GGC significantly, particularly on large problem sizes.

On the two smallest problem sizes, both collectors have no major collection and have similar numbers of minor collections. Since LGC has slightly higher allocation overhead, GGC is faster.

In sum, in a virtual memory environment in which a major collection incurs paging costs, LGC promotion strategy is preferable to GGC immediate promotion policy. By balancing the need to free up space in the top level and the need to keep objects there

until they die, LGC promotion strategy results in much fewer major collections than GGC immediate promotion policy, and hence shows superior virtual memory performance. On all except for the very small problem sizes, LGC significantly outperforms GGC.

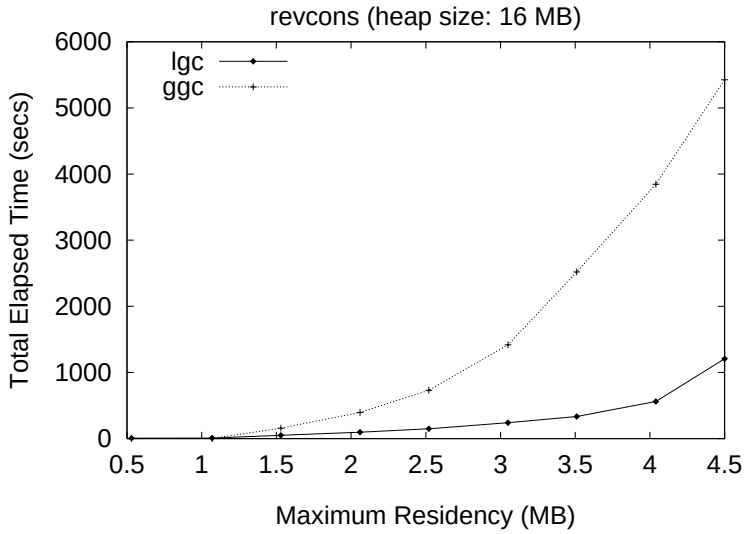

Figure 6: Paging performance of promotion policies: *revcons*

5.2.3 Large Programs Dominated By Long-Lived Objects

The examples so far have shown that on large problem sizes, LGC promotion strategy leads to substantially better performance than GGC immediate promotion policy. However, for problems that are dominated by long-lived objects, GGC immediate promotion policy may outperform LGC promotion policy. For such problems, keeping some of the long-lived objects in the top level brings no advantage, since they won't die and get reclaimed in a minor collection, and their longer stay in the top level will only force more frequent minor collections. GGC immediate promotion policy can quickly promote these long-lived objects to the bottom level, and hence avoid the repeated tracing cost that is incurred by LGC promotion policy. The following performance measurement shows an example of such a problem, so we can gain more insight into the performance of both promotion strategies.

The benchmark *saturated* is used in this experiment. It simply creates a long list by adding new items at the head, and never looks at the list again. Then, the list is discarded. The whole process is repeated several times. This benchmark is dominated by long-lived objects.

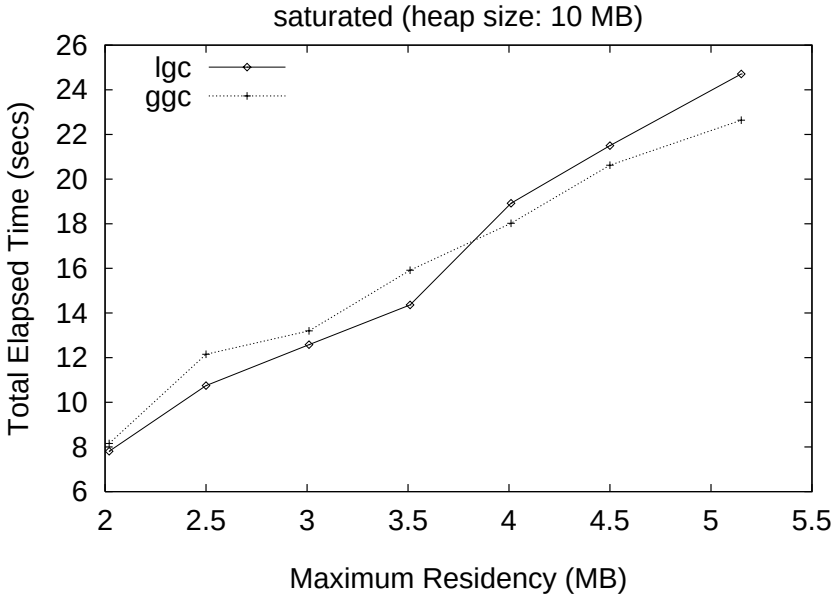

Figure 7: GGC may win on large programs dominated by long-lived objects

Figure 7 compares the performance of the two promotion policies on medium to large problem sizes of the benchmark *saturated*. A heap of 10 MB is used, and hence no paging cost is involved. In this figure, $F = 0.5 \cdot S$ for LGC.

6 Related Work

Recently, Clinger and Hansen proposed a nonpredictive generational collector that doesn't keep track of objects ages [5]. However, it is used for collecting objects in the oldest generation only. Also, it makes no effort to achieve better paging performance than conventional generational collectors. On their web pages, the authors don't claim performance improvements over the conventional generational collectors. Our LGC is also nonpredictive, but has significantly better performance than generational garbage collectors on several examples.

We recently learned that the latest test version 1.09 of SMLNJ uses 7-generational GGC with immediate promotion at the top level (or the youngest generation). One motivation for multigenerational GGC is to slow down promotion. We leave comparison of LGC to multigenerational GGC to future research. The best LGC will probably have fewer levels than the best multigenerational GGC. It appears that LGC will still have an advantage in avoiding larger collections balanced against GGC's advantage in allocation overhead. GGC's write barrier may become more costly due to the larger

number of levels. Whether GGC gains by making smaller top level collections than LGC, or loses by making more of these collections, will depend on the longevity of nodes in the particular benchmarks.

7 Conclusion

We have presented a new garbage collection algorithm which uses a heap structure and collection scheme similar to those of generational collectors but whose effectiveness doesn't rely on the age heuristic needed by the generational garbage collectors. We described a detailed design of this new garbage collection system. Through performance comparison with a generational collector with an immediate promotion policy, we showed that our new garbage collection algorithm not only substantially outperforms generational collectors with immediate promotion policy on several examples, but also demonstrates superior virtual memory performance.

8 Acknowledgement

We thank the anonymous referees for helpful comments that improved the presentation.

References

[1] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2), February 1989.

[2] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced language Implementation*, pages 89–100. MIT Press, Cambridge, Massachusetts, 1991.

[3] Henry G. Baker. Infant mortality and generational garbage collections. *ACM SIGPLAN Notices*, 24(4), April 1993.

[4] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11), November 1970.

[5] William D. Clinger and Lars T. Hansen. Generational garbage collection and the radioactive decay model. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Languages Design and Implementations*, June 1997.

[6] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Survey*, 13(3), September 1981.

[7] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4), October 1983.

[8] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage collector for virtual-memory computer systems. *Communications of the ACM*, 12(11), November 1969.

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

[10] Barry Hayes. Using key object opportunism to collect old objects. In *Proceedings of the 1991 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1991.

[11] Henry Lieberma and Carl Hewitt. A real-time garbage collector based on lifetimes of objects. *Communications of the ACM*, 26(6), June 1983.

[12] John McCarthy. Recursive functions of symbolic expressions and their computations by machine. *Communications of the ACM*, 3(4), April 1960.

[13] David Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, August 1984.

[14] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. In *Proceedings of the 1993 ACM Conference on Functional Programming Languages and Computer Architecture*, May 1993.

[15] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8), August 1967.

[16] Guanshan Tong. *Leveled Garbage Collection For Automatic Memory Management*. PhD thesis, University of Chicago, Chicago, IL, November 1997.

[17] David Ungar. Generation scavenging: A non-disruptive high performance storage management reclamation algorithm. In *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on practical Software Development Environments*, April 1984.

[18] John H. G. van Groningen. Optimizing mark-scan garbage collection. *The Journal of Functional and Logic Programming*, 1995(2), November 1995.

[19] Paul R. Wilson. Uniprocessor garbage collection techniques. *ACM Computing Survey*. To appear.

[20] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, May 1990.