Exploiting the Behavior of Generational Garbage Collector

Zhe Xu, Jia Zhao

I. Introduction

Garbage collection is a form of automatic memory management. The garbage collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program.\(^1\)

Comparing to explicit memory management, the advantages of garbage collection: it frees the programmer from dealing with memory deallocation, so the dangling pointer bugs, double free bugs and certain kinds of memory leaks will be eliminated. It also improves the mutator program’s spatial locality by relocating data used together in memory.

The motivation of garbage collection is obvious. However, the cost of gc is significant: Garbage collection use computing resources and will cause a pause time in the program run time. Also the time when a major collection will come is unpredictable, which may lead to very bad user experience.

The compiler creates and manages a run-time environment in which it assumes its target programs are being executed. Memory management is one of the important component that the compiler should deal with in run time environment.

The debate of whether using garbage collection or explicit memory management is still a hot topic. Garbage collection is an important productivity feature in most modern programming languages, such as Java and Python. As these programming languages becoming more and more popular, the performance of garbage collection is the bottleneck we want to break through.

Since performance matters in the garbage collection, our project explores how different parameters in garbage collection affect the performance. Also, our project designs several benchmarks that support or oppose the hypothesis that modern garbage collector made.

II. Related Work and Algorithm Design

A. Tracing Garbage Collector

Mark and sweep is a typical tracing garbage collector. Mark and sweep usually use a free list allocator, when the heap is full, it triggers a collection. During the collection, the collector traces and marks all the live objects, then scan all the objects, reclaim the unmarked objects and reset the marked objects.

The time complexity of this method is proportional to the number of allocated objects. Its maximum time latency is big and not practical in real-time systems.

\(^1\) http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29
B. Reference Counting Collector

Reference Counting is another typical garbage collector. It uses a reference counter on each object to trace its reachability. When the reference counter of an object reaches zero, which means there is no pointers pointing to it, the object becomes a garbage and will be collected by the collector.

However, it has been proven that this collector could not deal with the cycle link situation. When objects become cycle-linked by each other, even when they lose all the root pointers(pointers which directly from static memories), their reference counter will not be zero. In this case, the collector will fail to reclaim these resources.

C. Generational Garbage Collector With Aging

In our project, we will implement the generational algorithm with aging. It is the garbage collector used by Java and shows good performance for most real life programs and data sets.

The generational algorithm is based on the hypothesis that young objects die quickly and old objects survive a higher rate than young. There are many real life examples support this hypothesis, such as the recursion implementation of a program, the stack data structure both show this hypothesis is true. However there are some counter examples make this hypothesis doesn't hold. If a program mainly uses a FIFO queue based data structure, then it breaks this hypothesis.

As figure 1 shows, we divide the heap into 3 generations. The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this triggers a minor garbage collection. A young generation is collected very quickly. The surviving objects are aged and will eventually be moved to the old generation.

![Figure 1 Heap Structure](http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html)

The **Old Generation** is used to store long surviving objects. Also, a threshold is set for young generation object and when that age(i.e. number of times an object has been copied) is met,
the object gets moved to the old generation. Eventually the old generation needs to be collected. This triggers a major garbage collection which may cause a long latency.

The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here, too.

### III. Methodology

We would design a simulator to implement the generational garbage collector and analyze its performance with C++ language. Our simulator will read commands from stdin to create objects and delete links.

We would generate a serie of memory allocation commands according to or against the hypothesis we mentioned above. We will examine the model of real life program, and try to generate the commands follow the real data structure and run time features.

**Expected Test Data:**
1. Support hypothesis, i.e. young objects die quickly and old objects survive longer
2. Oppose hypothesis, i.e. data structures such as queue

We would implement the generational algorithm with two generations, the old generation and the young generation. The young generation will be further divided into three spaces for minor garbage collection(figure 2). Each time an object change a space in the young generation, the age will increment, when the age is over a threshold or the survivor space is full, the object will be copied to old generation.

![Figure 2 Copy live objects to survivor space](http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html)

We would test how the following parameters affect the performance of the garbage collection.
1. Age threshold, that is when to copy the object to the old generation.
2. How various young generation memory space size affect the garbage collection frequency?

With different parameters, we want to figure out how these commands will affect the performance of the generational garbage collector.

Finally, we generate some figures to show the test results.

---

IV. Implementation

A. Input Generation

The memory allocation generator can simulate memory allocate and deallocate instructions. In each iteration, there are three possible operations: Allocate a new node, add a link between two exists nodes and delete a link. We set probabilities for these three operations and use random number generator to determine which operation will take in each iteration.

Adaptive Delete Operation Percentage. The percentage of deleting operation is adaptive to the number of live nodes. When the total number of live nodes increases, the probability of deleting operation grows. As figure 3 shows, in the implementation, the starting deleting operation percentage is 40%. The rate grows about 5% with every 1000 more live nodes.

![Figure 3 Adaptive deleting rate](image)

The generator can generate memory allocates data support the hypothesis. In the delete operation, young objects have higher probabilities of being deleted. In the implementation, the young objects die rate is around 90%. There are some default settings for the generator: The default total allocated data can not exceed 4GB. The total live nodes can not exceed 5000.

The generator can also generate data that opposes the hypothesis. It can generate memory usage like queue data structure. That is deleting the oldest objects first. With this set of data, we could test the worst case behavior of the garbage collector.

B. Garbage Collector Implementation

The Garbage Collector contains three major parts(Figure 4): the controller which implements the main algorithm of the garbage collection, the memory pool object which simulates the main memory and proceeds memory allocation and deletion requests from the controller and the objects class which contains reference counter and mark bit and simulates the real objects created by the top user.
The controller accepts three types of system requests in two command formats as following:
create(parent, child, size)
delete(parent, child)

Create Objects/Links Requests. The first one is creating new objects. This request will allocate memory to the user and link the new object to its parent. The second request is linking object which simulates adding a reference or a pointer to the target object. Both of the previous two requests share the same command format, that is “create(parent, child, size)”. If the child does not exist in the system, then it is a creating request and the size parameter will be considered to allocate memory to the new object. Otherwise, it is a linking request and the garbage collector will only add a link from the parent to the child and ignore the size parameter.

Delete Link Requests. The last request is the deleting link request, which simulates deleting a reference or a pointer to an object. This request uses the command “delete(parent, child)”. To simplify the interface, these is no deleting object request in the whole simulation. All the stack or static object deleting will be translated to deleting all their outgoing links.

Each time, the controller received a “create” command, it will check whether the child exists or not. If not, it will first create this object and allocate memory space from EDEN memory and then link these two objects together. When the EDEN space is out of memory, the controller will do a minor garbage collection on the EDEN space and the current Survivor space.

Minor Garbage Collection. The minor collection uses reference counting algorithm and only deletes objects with no reference, therefore it can be done very quickly. After this minor garbage collection, the controller will move all the living objects in these two space to the standby Survivor pool and increase their age by 1. If the age of an object is exceeding the age threshold, the controller will move it to the old space.
**Major Garbage Collection.** If the old space is out of memory during this movement, a major garbage collection will take place. This procedure uses mark and sweep algorithm and will clean all the unreachable objects.

![Garbage Collector Process Diagram](image)

**Dynamic Adjust Age Threshold.** Besides, the collector makes dynamic adjustment on the age threshold. During the simulation, the controller may receive dense creating requests without deleting links. If this happens, the Survivor memory space may be not enough to wait for the objects to reach the threshold. If this happens, the controller will decrease the age threshold and move objects to the Old space earlier than the parameter indicates.

### V. Data Analysis

Using the runTest.py script, we could generate three sets of output. The first set is how different heap sizes affect the GC behavior; the second set is how different young memory space sizes influence the performance; the third set is how various dynamic age thresholds help GC work.

**A. Collector Sensitive to Heap Size**

In this test, we gave the garbage collector four different heap settings, see Table 1.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Total Memory</th>
<th>Young Memory</th>
<th>Survivor Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting 1</td>
<td>4G</td>
<td>2G</td>
<td>640M</td>
</tr>
<tr>
<td>Setting 2</td>
<td>2G</td>
<td>1G</td>
<td>320M</td>
</tr>
<tr>
<td>Setting 3</td>
<td>1G</td>
<td>512M</td>
<td>160M</td>
</tr>
<tr>
<td>Setting 4</td>
<td>512M</td>
<td>256M</td>
<td>80M</td>
</tr>
</tbody>
</table>

Table 1 Different Heap Settings
Figure 6 Average Minor GC frequencies at Different Heap Size

It is clear that decreasing heap size tends to increase the frequency of garbage collection. As figure 6 shows, we count the average minor garbage collection frequencies at different heap size. Since we use generational garbage collection, the data does not trigger the major GC for most of the test cases, therefore the simulation only count the minor GC frequencies.

Although with larger heap, GC collects less frequently, the amount of memory collected in each time is larger, thus causes larger pause. Also the memory resource is limited, a better choice is to use moderate heap size and adjust the young memory space size as the following testing shows.

**B. Young generation size affects Collector**

In this simulation, we give the collector different young space size as table 2 shows.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Total Memory</th>
<th>Young Memory</th>
<th>Survivor Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting 1</td>
<td>1G</td>
<td>640M</td>
<td>200M</td>
</tr>
<tr>
<td>Setting 2</td>
<td>1G</td>
<td>512M</td>
<td>160M</td>
</tr>
<tr>
<td>Setting 3</td>
<td>1G</td>
<td>256M</td>
<td>72M</td>
</tr>
<tr>
<td>Setting 4</td>
<td>1G</td>
<td>128M</td>
<td>32M</td>
</tr>
</tbody>
</table>

Table 2 Different Young Space Settings
As figure 7 shows, the larger the young generation size, the less frequent that minor garbage collection occurs, however the cost for each minor GC is more. Here we simulate the cost using the average number of megabytes copied in the GC. It is not accurate since each collection has a fix overhead, but generally the memory copy consumes the majority of time in GC.

If we calculate the total cost by multiplying the GC frequencies and average time consumed together, we can see the trend is decreasing at first, then increasing (See figure 8). As we predicts, largest or smallest young space size are not best, moderate young memory space size could gain better performance in the measure of total time.

A change in young space size will trigger the collection at different points. Consider a program has relatively short live data structures. Using a small young space(eden space) size, it could trigger minor GC in the middle of the program building the data structure; and in a slightly larger young space size, minor GC could happen after the data structure dies. In the second case, we do not have much to copy and the GC time is lesser. That is why we need moderate young space size to achieve best performance and small pause in GC.
C. Dynamic Adjusting Age Threshold
Since the garbage collector could dynamic adjust the age threshold according to the input, we expect the testing result to be similar with different initial threshold.

In the simulation, we change the initial age threshold as 2, 4, 6, 8. As we expected, the frequencies of GC and the average byte copied each time is almost the same.

D. Worst Case Behavior
If we run the simulation with memory allocation like queue data structures. It will trigger much more major collections. (See table 3) It is easy to understand because we always delete old objects and never delete new objects.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Total Memory</th>
<th>Young Memory</th>
<th>Survivor Space</th>
<th>Worst Case Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting 1</td>
<td>1G</td>
<td>640M</td>
<td>200M</td>
<td>8 Minor, 5 Major, Memory Limit Exceed</td>
</tr>
<tr>
<td>Setting 2</td>
<td>1G</td>
<td>512M</td>
<td>160M</td>
<td>9 Minor, 3 Major, Memory Limit Exceed</td>
</tr>
<tr>
<td>Setting 3</td>
<td>1G</td>
<td>256M</td>
<td>72M</td>
<td>25 Minor, 6 Major</td>
</tr>
<tr>
<td>Setting 4</td>
<td>1G</td>
<td>128M</td>
<td>32M</td>
<td>44 Minor, 5 Major</td>
</tr>
</tbody>
</table>

Table 3  Worst Case Behavior with a Queue Data Structure

With this worst case scenario, we could see this generational garbage collector highly depend on the hypothesis that young objects should die with a high probability. If the hypothesis does not hold, the behavior of the generational garbage collector is conservative.

VI. Conclusion

This project examines the behavior of generational garbage collector on different choices of heap size, young generation size and age threshold. A few key observations emerge. First, the choice of young generation size should consider both the collection frequency and the average collection time, thus lead to a first decreasing, then increasing curve in total collection time. Second, dynamic adjust age threshold could adapt to different data behavior better. Last, if the program does not follow the generational hypothesis, the collector tend to trigger major garbage collection more often and the total performance is not ideal. These results could guide the programmer to use right collector for their program and offer some insights for memory management.

VII. Bibliography