QUANTITATIVE ANALYSIS OF BOEHM'S GC

Guan Xuetao, Zhang Yuanrui, Gou Xiaogang and Cheng Xu
MicroProcessor R&D Center of Peking University, Post Code: 100871, Beijing, China

ABSTRACT

The term *garbage collection* describes the automated process of finding previously allocated memory that is no longer in use in order to make the memory available to satisfy subsequent allocation requests. We have reviewed existing papers and implementations of GC, and especially analyzed Boehm's C codes, which is a real-time mark-sweep GC running under Linux and ANSI C standard. In this paper, we will quantitatively analyze the performance of different configurations of Boehm's collector subjected to different workloads. Reported measurements demonstrate that a refined garbage collector is a viable alternative to traditional explicit memory management techniques, even for low-level languages. It is more a trade-off for certain system than an all-or-nothing proposition.

KEYWORDS: conservative, garbage collection, incremental, mark-sweep, real-time

1 INTRODUCTION

Garbage collection is a hotspot at the end of the eighties and the beginning of the nineties in the twentieth century, and most papers have emerged for various collection strategies.

Traditional techniques include reference counting, mark-sweep and copying [6]. And many variations are possible. Advanced techniques have been developed to ameliorate the situation in two different ways: first, by splitting the work traditionally performed in a single collector invocation into finer-grained parts; second, by exploiting statistical observations on object lifetimes. Generational collection, hardware-assisted collection and incremental garbage collection are all advanced choices. Generally, different collectors show different advantages and disadvantages.

Garbage collection is not an all-or-nothing proposition. It is a trade-off for certain system. On one hand, it can reduce the cost of development of a software system and the occurrence of dynamic memory management errors in both prototype and production components. It can offer storage throughputs that exceed the capabilities of traditional memory management techniques and enhance the software modeling and reusability. But, on the other hand, it may increase the complexity and reduce the overall system performance. And in the interactive and real-time system, it may suspend applications periodically.

As is known to all, performance is essential to system software and its improvement needs expenses of space, time or hardware. Many papers have showed the performance comparison between systems with certain collection and one with C's traditional malloc and free though the comparison is considerably dependent on the underlying system and the relevant benchmarks.

In this paper, we report the quantitative performance analysis of several different configurations of Boehm's incremental mark-sweep garbage collection system subjected to several different workloads. Reported measurements demonstrate that a refined garbage collector is a viable alternative to traditional explicit memory management techniques, even for low-level languages like ANSI C.

2 BOEHM'S REAL TIME COLLECTOR

For real-time computer systems, high reliability and fault tolerance of both software and hardware are of utmost importance. Real-time software engineer is more concerned with ensuring the system's reliability than optimizing its throughput.

To meet real-time system requests, the GC research community has put forth a lot of proposals for real-time garbage collection. The real-time engineer must be able to derive the worst-case execution time for each task in the system. Tasks interact with the garbage collector by allocating new memory and by fetching and storing to memory locations contained within heap-allocated objects. The delays associated with memory allocation and memory access must be bounded by constants that are small enough to allow the real-time schedule for the complete system to be analyzed or pre-computed. If the garbage collector runs on CPU, its computation must be modeled as a periodic task with bounded execution time and a fixed period of execution. The performance impact of garbage collection on the complete system must be small enough that it does not prevent system from meeting its real-time deadlines.

Boehm's collector is a real-time collector. It mainly uses mark-sweep algorithm to reclaim garbage produced
by user or system, and do allocations dynamically.

This collector has two important features: conservative and incremental. It can operate with only minimal information about the layout of the client program's data. Instead of relying on compiler provided information on the location of pointers, they assume that any bit pattern that could be a valid pointer is a valid pointer. It can also take care to interleave collections with the main computation, in order to avoid distracting pauses for both real-time and interactive applications.

Conceptually, Boehm's collector operates roughly in four phases: **Preparation phase** clears all mark bits, indicating that all objects are potentially unreachable. **Mark phase** marks all objects that can be reachable via chains of pointers from variables. **Sweep phase** scans the heap for inaccessible, and hence unmarked, objects, and returns them to an appropriate free list for reuse. **Finalization phase** inserts unreachable objects into the queues, which have been registered for finalization outside the collector.

### 3 PERFORMANCE OF BOEHM'S GC

Early versions of this collector were developed as a part of research projects supported in part by the National Science Foundation and the Defense Advance Research Projects Agency. Much of the code was rewritten by Hans-J. Boehm, at Xerox PARC.

#### 3.1 Benchmark Selection

Recently, we have ported several allocation-intensive C programs to our garbage-collected system. The following table is the description of the benchmarks.

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groff</td>
<td>1.17.2</td>
<td>GNU groff - front end for the groff document formatting system. Normally it runs the tgif program and a post processor appropriate for the selected device. Called Subprograms: troff version 1.17.2, grops version 1.17.2.</td>
</tr>
<tr>
<td>Bzip2</td>
<td>1.0.1</td>
<td>A block-sorting file compressor. It compresses files using the Burrows-Wheeler block sorting text compression algorithm, and huffman coding. Generally compression is considered better than more conventional LZ77/LZ78-based compression. Performance is close to the PPM family of statistical compression.</td>
</tr>
<tr>
<td>Sort</td>
<td>2.0.14</td>
<td>Belongs to textutils 2.0.14. Sort lines of text files.</td>
</tr>
<tr>
<td>Diff</td>
<td>2.7.2</td>
<td>Belongs to diffutils 2.7.2. Find differences between two files.</td>
</tr>
<tr>
<td>Pei</td>
<td>1.254</td>
<td>Simulates the detection and de-screening of Halftone illustrations algorithm, it takes grey-scale images as inputs and produces a de-screened grey-scale output image for each halftone illustration detected.</td>
</tr>
<tr>
<td>Eppstein</td>
<td>1.1</td>
<td>Implementation of Eppstein's Algorithm that enumerates (by increasing weight) the K shortest paths in weighted graphs.</td>
</tr>
</tbody>
</table>

In order to analyze these programs in our environment, it is necessary to:

1. Add collection's library into the compiling libraries of certain program. Also, the program should include collection's header.
2. Replace all occurrences of malloc with appropriate invocations of new.
3. Replace all occurrences of free with delete.

#### 3.2 Benchmark Allocation Statistics

The following table shows the behavior of each benchmark Malloc.

<table>
<thead>
<tr>
<th>Program</th>
<th>Bzip2</th>
<th>Chksum</th>
<th>Diff</th>
<th>Eppstein</th>
<th>Groff</th>
<th>Pei</th>
<th>Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malloc Times</td>
<td>11</td>
<td>156</td>
<td>115</td>
<td>20</td>
<td>9.94E5</td>
<td>7544</td>
<td>126</td>
</tr>
<tr>
<td>Malloc bytes mean</td>
<td>6.84E5</td>
<td>58</td>
<td>142</td>
<td>4026</td>
<td>18</td>
<td>342</td>
<td>5.28E6</td>
</tr>
<tr>
<td>Malloc bytes max</td>
<td>3.60E6</td>
<td>1024</td>
<td>4096</td>
<td>40000</td>
<td>24576</td>
<td>80523</td>
<td>1.84E7</td>
</tr>
<tr>
<td>Malloc bytes min</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Malloc bytes sum</td>
<td>7.52E6</td>
<td>9113</td>
<td>16358</td>
<td>80523</td>
<td>1.84E7</td>
<td>2.58E6</td>
<td>2.45E7</td>
</tr>
<tr>
<td>Malloc variance</td>
<td>2.08E12</td>
<td>20520</td>
<td>3.31E5</td>
<td>1.15E8</td>
<td>2495</td>
<td>5.28E6</td>
<td>4.74E12</td>
</tr>
<tr>
<td>Malloc time mean(ns)</td>
<td>9.72</td>
<td>3.88</td>
<td>3.80</td>
<td>6.52</td>
<td>3.76</td>
<td>3.94</td>
<td>3.74</td>
</tr>
<tr>
<td>Malloc time max(ns)</td>
<td>21</td>
<td>53</td>
<td>15</td>
<td>17</td>
<td>14487</td>
<td>82</td>
<td>18</td>
</tr>
<tr>
<td>Malloc time min(ns)</td>
<td>2.98</td>
<td>2.98</td>
<td>2.98</td>
<td>2.98</td>
<td>2.98</td>
<td>2.98</td>
<td>2.98</td>
</tr>
<tr>
<td>Malloc time variance</td>
<td>56</td>
<td>18</td>
<td>4</td>
<td>17</td>
<td>485</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

These benchmarks show distinguished behaviors of the memory allocations. Bzip2 does a very little malloc, but allocating a very large memory each time and the variance of the malloc bytes is also big. Groff does the most frequent allocations and the required memories are always small. The variance of its malloc bytes is the smallest, but the variance of its malloc time is the biggest. Cksum, Diff, and Sort represent for the programs, whose malloc times are of a little difference, but the memory requirements are distinct, from low to high. Eppstein does not do much malloc, while Pei does frequent malloc. But the variance of their memory requirements is small, respectively.

The malloc time in the table is measured without garbage collection. And they are all within 10ns, which is very short. The least malloc time in the measured system takes roughly 3ns.

#### 3.3 Comparison Between GC and NonGC

The allocator of traditional explicit memory management is distributed as part of the GNU libc distribution. This allocator uses two different allocation algorithms, depending on the size of the requested memory. Programs with GC will use more time and more space.

The following shows program execution time for GC and NonGC.
Table 3 Program execution time of GC and NonGC

<table>
<thead>
<tr>
<th></th>
<th>Bzip2</th>
<th>Diff</th>
<th>Eppstein</th>
<th>Pei</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>2.82</td>
<td>0.05</td>
<td>2.99</td>
<td>0.06</td>
</tr>
<tr>
<td>System</td>
<td>3.09</td>
<td>0.03</td>
<td>2.77</td>
<td>0.00</td>
</tr>
<tr>
<td>Wall</td>
<td>0.02</td>
<td>0.04</td>
<td>0.16</td>
<td>0.00</td>
</tr>
<tr>
<td>User</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>System</td>
<td>0.06</td>
<td>0.64</td>
<td>0.76</td>
<td>0.32</td>
</tr>
<tr>
<td>Wall</td>
<td>0.30</td>
<td>1.18</td>
<td>4.13</td>
<td>3.30</td>
</tr>
</tbody>
</table>

User indicates user time, system is the system time and wall is the total time. Program with garbage collection takes a little longer than without GC. For example, Diff uses 0.02 s to do malloc and uses 0.05 to do malloc with GC.

Here we have program size of GC and NonGC.

Program size comparison: difference mean = 41443 bytes

Table 4 Program size of GC and NonGC

<table>
<thead>
<tr>
<th></th>
<th>glibc</th>
<th>gc</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>89049</td>
<td>129690</td>
<td>40641</td>
</tr>
<tr>
<td>diff</td>
<td>251589</td>
<td>293978</td>
<td>42389</td>
</tr>
<tr>
<td>eppstein</td>
<td>26354</td>
<td>67159</td>
<td>40845</td>
</tr>
<tr>
<td>peiConvolve</td>
<td>317430</td>
<td>359035</td>
<td>41605</td>
</tr>
<tr>
<td>peiExtract</td>
<td>144744</td>
<td>186349</td>
<td>41605</td>
</tr>
<tr>
<td>peiMake</td>
<td>147422</td>
<td>188995</td>
<td>41573</td>
</tr>
</tbody>
</table>

This table shows the program size with GC. Obviously, the size becomes larger than the traditional malloc. The difference mean is 41443 on average.

3.4 Behavior of Boehm's Collector

We have following configures here with initial their values:

- STUBBORN_ALLOC
  - Define stubborn allocation primitives
- FIND_LEAK
- ALL_INTERIOR_POINTERS
- MERGE_SIZES
  - Round up some object sizes, so that fewer distinct
- MINHINCR = 16
  - Minimum heap increment, in blocks of HBLKSIZE
- MAXHINCR = 512
  - Maximum heap increment, in blocks
- TIME_LIMIT = 50
  - pause times
- GC_incremental = 0
  - Using incremental collection
- GC_full_freq = 4
  - Number of partial collections between full collections
- GC_quiet = 0
  - Disable statistics output
- GC_word GC_free_space_divisor = 4

We try to make sure that we allocate at least N/GC_free_space_divisor bytes between collections, where N is the heap size plus a rough estimate of the root set size. Increasing its value will use less space, but more collection time. Decreasing it will appreciably decrease collection time at the expense of space. GC_free_space_divisor = 1 will effectively disable collections.

The following configurations are all compared with benchmarks.

1. Findleak: #undef FIND_LEAK
2. Noninter: #undef ALL_INTERIOR_POINTERS
3. Nonstub: #undef STUBBORN_ALLOC
4. Heapinc8: #define MINHINCR 16
5. Heapinc32: #define MINHINCR 16
6. Time60: #define TIME_LIMIT 60
7. Time80: #define TIME_LIMIT 80
8. Freesp2: GC_free_space_divisor = 2
9. Freesp3: GC_free_space_divisor = 3
10. Freesp5: GC_free_space_divisor = 5
11. Freesp6: GC_free_space_divisor = 6
12. Freesp7: GC_free_space_divisor = 7
13. Freesp8: GC_free_space_divisor = 8
14. Increment: GC_incremental = 1
15. Freq2: GC_full_freq = 2 & GC_incremental = 1
16. Freq3: GC_full_freq = 3 & GC_incremental = 1
17. Freq5: GC_full_freq = 5 & GC_incremental = 1
18. Freq6: GC_full_freq = 6 & GC_incremental = 1
19. Freq7: GC_full_freq = 7 & GC_incremental = 1
20. Freq8: GC_full_freq = 8 & GC_incremental = 1

And we only choose four programs to compare: bzip2, diff, pei and eppstein.

Statistic1: max heap size for each configuration

The following four pictures describe the max heap size of each program during garbage collection. FREQ reflects the heap increasing times, and heap size is the multiple times of 65536. For example, benchmark has heap size 64, which is actually 64*65536 bytes.

![Fig. 1 Max heap size for bzip2](image1)

![Fig. 2 Max heap size for diff](image2)
From tables, we know that if the collector allows incremental garbage collection, then the heap will become about 2-5 times larger than the heap with world-stop-collection. In the incremental mode, the collector reclaims less garbage than in the stop-the-world mode, so it makes heap grow faster. The configuration freq3, freq5, freq6, freq7 and freq8 are all in incremental mode. The result is most obvious in program eppstein.

If the pause time increases to 80 ms, the heap also increases. For the upper time increases, the marking time may also increase to push more objects in the mark stack. So failure to finish a collection may be increased. This causes heap to grow lager.

When GC_free_space_divisor increases, the frequency of collection is also increased, but each time, the collector may not collect enough areas for allocation needs, thus, expanding heap a little as a result in program eppstein.

Statistic2: collection times

We look at program bzip2 and diff under configuration freq2. The general program may trigger 5 times of full mark, and the last column is the average bytes allocated between each invocation.

Statistic3: reclaimed bytes

This diagram mainly shows the reclaimed bytes during each collection. In the picture, the negative number does not really mean that it is smaller than zero, but just a little bigger than zero. This indicates that we reclaim just a few areas each time, which may not satisfy the applying of allocation.

Statistic4: object size distribution

This measurement is on the benchmark. In most cases, an object's size is smaller than 400 bytes, and a few objects are larger. This reflects the original size that required by the user. In the collector, we actually merge size first and align it before the allocation takes place. And this can reduce the distinctions of objects' size, thus saving spaces.

Statistic5: wasted areas

We know that the actually object size is larger than the original one for merging and alignment reasons. So each allocation can produce wasted areas. There is a phenomenon, that the average allocation size is larger, the wasted area size is smaller. The malloc bytes mean of bzip2 is 683989, and its max wasted area is 4732 bytes. The malloc bytes mean of diff is 142.2324, the smallest, but its max wasted area can be up to 11536...
bytes. The malloc bytes mean of pei is 342.0405 and its wasted areas are mostly small. And eppstein has the smallest wasted areas. Compared with program's malloc and free behavior analysis, we could conclude that more allocations means more wasted bytes.

Statistic6: time measurements

Collection time is the time of a whole garbage collection. The result is that it mainly takes 20ms to finish this. And the initial time for preparation phase takes less than 10ms. Some times finalizer is evoked, and it also takes less than 10ms. They are all with 50ms defined in benchmark.

Statistic7: object kinds comparison

Boehm's collector is a conservative garbage collector, which can operate with only minimal information about the layout of the client program's data. Instead of relying on compiler provided information on the location of pointers, they assume that any bit pattern that could be a valid pointer is a valid pointer. Generally, this is safe only under the assumption that objects do not move.

It uses a two-level tree data structure to aid in fast pointer identification. Two kinds of objects in the collector are called pointer-free object and normal object respectively. Pointer-free object is also called atomic object, which contains no pointers. Normal object is also called composite object, which contains pointers. The following two pictures are comparisons between atomic objects and composition objects in use. The picture reflects the result of program bzip2. In the program, the atomic objects are nearly 0 bytes, meaning that most objects contains pointers. This may produce overhead for conservative garbage collector to check the candidate pointers.

Statistic8: false pointers measurement

Boehm's collector maintains two different kinds of blacklisting. A page may be black listed for interior pointer references if it was the target of a near miss from a location that requires interior pointer recognition, e.g. the stack. If the near miss came from a source that did not require interior pointer recognition, it is black listed with normal. A page black-listed in this way may appear inside a large object, so long as it is not the first page of a large object.

The configuration is noninter. False pointers from the stack are more dangerous, but they are not many, luckily. The following table shows the fact.

<table>
<thead>
<tr>
<th></th>
<th>Bzip2</th>
<th>Diff</th>
<th>Eppstein</th>
<th>Pei</th>
</tr>
</thead>
<tbody>
<tr>
<td>False pointers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>without interior pointer</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td>63</td>
</tr>
<tr>
<td>False pointers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>from the stack</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
4 CONCLUSIONS

Now, there is no pure-mono-GC, i.e., any modern GC is the integration of several GC mechanisms and some tricks. Every method is only an idea inside the real GC.

As discussed above, we have known the features of the garbage collectors. We can't imagine a collector have all advantages of others, so we must make trade-off. This trade-off is based with the target, or the performance to which we attach great importance. A platform can evaluate some performance, but for the concrete system, such as embedded system and hardware-assisted garbage-collected system, it can't describe the distinguishing parts, or can't satisfy the requirements, then the results are questionable, and it just possible is our future work.

Our future works are concentrating in the deeply analysis of the Boehm's source code and performance. We plan to do the collection research on our embedded system, and there are two directions, described as following:
1. Adding GCMM to the MMU, and implementing and analyzing hardware-assistant collectors, especially the real-time features.
2. Inserting collectors to the embedded operating system afforded with the compiler, and analyzing the overall performance, especially the real-time features.

Both directions are practical works, and will form the indeed useful materials.

REFERENCES