Generational Collection: Efficiency in Memory Reclamation

Shubhnandan S. Jamwal
PG Department of Computer Sciences, University of Jammu
jamwalsnj@gmail.com

ABSTRACT
The issue of memory reclamation is the most important aspect of garbage collection. Its role becomes important in the devices with limited amount of the memory. Most of the digital devices having large memory such as personal computers do not have the limitation of memory. The practical obstacle is in implementing the OOPS with garbage collectors in devices with small memory such as mobile phones. Since the inception of the concept of the memory reclamation the different garbage collector algorithms are designed and implemented. In this paper the issue of the memory reclaimed by the garbage collector is studied with respect to the multithreaded programming. The memory reclaimed by the garbage collector i.e Serial (SR GC), Parallel (PR GC), Incremental (INC) and Generational Garbage (TR) collectors is studied. It is observed that in all the stack sizes for threaded application the memory reclaimed by the parallel collector is 77.83%, 22.65% and 17.09% more than the serial collector, generational collector and Incremental collector where as the in heavily threaded applications the Incremental collector 47.19%, 76.54% and 33.48% reclaims more memory than the serial, parallel and the generational collector respectively.

KEYWORDS
Garbage collector, Mutator, Serial, Parallel, Incremental, Generational, Train, Stack, Multithreaded

1. INTRODUCTION
Garbage Collection is a technique of automatic reclamation of allocated program storage and was first proposed by McCarthy [1]. Other techniques for storage reclamation also exist and they are explicit programmer-controlled reuse of storage used in Pascal, C, and reference counting [2] and so on. Garbage collection plays an increasingly important role in next generation Internet computing and server software technologies. However, the performance of collection systems is largely dependent upon application execution behavior and resource availability [3]. When an object is no longer referenced by the program the heap space it occupies must be recycled so that the space is available for subsequent new objects. The Garbage collector must some how determine which objects are no longer referenced by the program and make available the heap space occupied by such unreferenced objects. In addition to freeing unreferenced objects, the garbage collector must also combat heap fragmentation. Heap fragmentation occurs through a course of normal program execution. New objects are allocated and unreferenced objects are freed such that the free blocks of the heap memory are left in between blocks occupied by live objects. Request to allocate new objects is to be filled by extending the size of the heap even though there is enough unused space available in the existing heap.

One of the main advantages of garbage collection is that, it relieves programmers from the burden of freeing allocated memory. Knowing when to explicitly free allocated memory can be very tricky. Giving this job to the compiler has many advantages. Second advantage of garbage collection is that the garbage collection ensures program integrity. Garbage collection is an important part of Java’s security strategy. Java programmers are unable to accidentally crash the finalize and free unreferenced objects on the fly, but the programmers in the garbage collected environment have less control over the scheduling of the CPU time devoted to freeing objects that are no longer needed.

2. REVIEW OF LITERATURE
Sunil Soman and Chandra Krintz [3] showed that application performance in garbage collecting languages is highly dependent upon the application behavior and on underlying resource availability. They further proved that given a wide range of diverse garbage collection algorithms, no single system performs best across all programs and heap sizes. They further presented a Java Virtual Machine extension for dynamic and automatic switching between diverse, widely used GCs for application specific garbage collection selection. Further they described a novel extension to extant on-stack replacement (OSR) mechanisms for aggressive GC specialization that is readily amenable to compiler optimization.

Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith [4] observed that when resources are abundant, there is no clear winner in application speed. However, when memory is limited, the hybrid collector (using mark-sweep for the mature space and semi-space copying for the nursery) can deliver at least 50% better application throughput. Therefore parallel collector seems best for online transaction processing applications.

Stephen M Blackburn, Perry Cheng, Kathryn S McKinley [5] observed that the overall performance of generational collectors as a function of heap size for each benchmark is mainly dictated by collector time. Semi Space is often the best in large heaps, but Mark Sweep does better in tight heaps. The overall results are not encouraging for constrained memory. Even with generational collectors, memory management costs are prohibitive. Garbage
collection algorithms still trade for space and time which needs to be better balanced for achieving the high performance computing.

Katherine Barabash, Yoav Ossia and Erez Petrank [6] presented a modification of the concurrent collector, which improves the throughput, the memory footprint, and the cache behavior of the collector without foiling the other good qualities (such as short pauses and high scalability). They implemented their solution on the IBM production JVM and obtained a performance improvement of up to 26.7%, a reduction in the heap consumption by up to 13.4%, and no substantial change in the (short) pause times. The modified algorithm was subsequently incorporated into the IBM production JVM.

Stephen M Blackburn, Perry Cheng and Kathryn S McKinley [7], experimental design reveals key algorithmic features and how they match program characteristics to explain the direct and indirect costs of garbage collection as a function of heap size on the SPEC JVM benchmarks. They find that the contiguous allocation of copying collectors attains significant locality benefits over free-list allocators. The reduced collection cost of the generational algorithms together with the locality benefit of contiguous allocation motivates a copying nursery for newly allocated objects. These benefits dominate the overheads of generational collectors compared with non-generational and no collection, disputing the myth that “no garbage collection is good garbage collection.” Performance is less sensitive to the mature space collection algorithm in our benchmarks. However the locality and pointer mutation characteristics for a given program occasionally prefer copying or mark-sweep.

Jurgen Heymann [8] presented an analytical model that compares all known garbage collection algorithms w.r.t. various performance measures. The overhead functions are easy to measure and tune parameters and account for all relevant sources of time and space overhead of the different algorithms.

2.1 ENVIRONMENT
The hardware used for conducting the tests was Intel core(R) Core(TM) 2 CPU T5600, 1.83 GHz, L1 cache of 64KB, L2 cache of 2048KB and 512 MB RAM. Microsoft Windows XP Professional Version 2002 Service pack 2 was used as an operating system. Java version "1.5.0" Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-b64), Java HotSpot(TM) Client VM (build 1.5.0-b64, mixed mode) is used as compiler.

3. BENCHMARKS AND METHODOLOGY
In this paper the benchmarks are of multithreaded in nature. It is worth observing that essentially all Java applications are multi-threaded, even those that only start with a single thread. The applications of the JAVA are multithread because the standard class libraries in the Java platform and the virtual machine itself start a number of system threads to handle object finalization and other background tasks. The performance of the garbage collectors is well understood in single threaded application and garbage collectors work extremely well. The garbage collectors impose no overhead on the performance of the mutator. Allocations have to check that enough space is free for the generations at any moment during the execution of mutator, and when this check fails garbage collectors is initiated. This ideal situation cannot be extended to multi-threaded systems in which multiple threads are executing at a time and context switching for the invocation of garbage collector is restricted to occur only at specific points. When the mutator enters into the critical section of the invocation of garbage collector the mutator is paused thus causing considerable loss of performance to the application. In this paper investigation, the benchmarks are categorized into two groups

- Threaded application
- Massively threaded application

The benchmarks for threaded applications is composed of more than twenty threads in each of the mutator and benchmark suite of massively threaded applications is composed of more than forty threads in each mutator. Threaded applications are composed of it_bs20t, it_ins20t, it_mer20t, it_sel20t, it_qs20t, it_shell20t and it_mix. Iterative constructs of bubble sort, insertion sort, merge sort, selection sort, quick sort and shell sort are included in first six programs. it_mix is composed of different iterative constructs of prime, armstrong, factorial etc. The sorting techniques sorts 50000 numbers which are generated as the random numbers by the Random class with the help of the following code

```java
Random r = new Random();
int array[] = new int[50000];
for(int i=0;i<50000;i++)
array[i] = r.nextInt();
```

The array of randomly generated numbers is passed to the appropriate sorting function. The calculation of time of the execution of the individual thread is calculated with the following lines of the code

```java
long beg = System.currentTimeMillis();
bubbleSort(array, array.length); // End Time
long end = System.currentTimeMillis();
```

beg and end variable records the starting and ending time of a particular thread. Each thread is allowed to finish on priority of its creation before the creation of other threads with the help join method. Then the average of all the threads of the application part is calculated.

The benchmark suite of massively threaded applications is composed of more than forty threads in each of the mutator. Massively threaded applications are composed of it_bs40t, it_ins40t, it_mer40t, it_sel40t and it_qs40t. This benchmark suite is also composed of iterative constructs of techniques which are discussed in application.

Each of the garbage collectors is activated during the execution of the benchmark. The four garbage collector i.e Serial, Parallel, Incremental and Generational Garbage collectors are activated by passing command line parameters to the JVM. The memory reclaimed by each of the collector after the execution of the
benchmark is recorded. Then the average of all the benchmarks is analyzed.

3.1 ISSUE OF MEMORY RECLAMATION
The memory reclamation is the most important aspect of garbage collection. Its role becomes important in the devices with limited amount of memory. Most of the digital devices having large memory such as personal computers do not have the limitation of memory. The practical obstacle is in implementing the OOPS with garbage collectors in devices with small memory such as mobile phones. The embedded systems have very less memory. Therefore, the issue of memory reclamation is important in these devices. The feature of more and more memory reclamation by the garbage collectors brings the benefits of high-level, object-oriented languages to the world of small wireless devices. There may be a separate version of JAVA available for these types of devices but still the issues of memory reclamation in all the systems is of utmost importance. In distributed application servers the issue of the memory reclamation remains hot spot.

4. RESULTS AND DISCUSSIONS
Each of the benchmark is executed in small stack sizes of 4mb, 8mb, 16mb, 32mb and 64mb. Each benchmark is executed five times and then the average of all the benchmarks is calculated for each of the garbage collector.

4.1 THREADED APPLICATIONS
The results in table 1 depict the average of five runs for each of the benchmark for the threaded applications. Figure 1 represents the pictorial representation of the performance of the garbage collectors in small stack sizes. In threaded applications, the memory reclaimed by the parallel garbage collector is very high particularly in footprints of 4mb to 32mb. The memory reclaimed by the serial collector is very less in footprints of 4mb to 32mb but the memory reclaimed by all the collectors in 64mb of stack becomes equal. The average in all the stack sizes for threaded application reveals that the memory reclaimed by the parallel collector is 77.83% more than the serial collector, 22.65% more than the generational collector and 17.09% more than the Incremental collector.

4.2 MASSIVELY THREADED APPLICATIONS
The performance of the Incremental collector in massively threaded application is far better than generational

| Table 1. Average memory reclaimed by all garbage collectors in threaded applications |
|-------------------------------------------------|--------|-------|-------|-------|
|        | SR GC  | PR GC  | TR GC  | INC GC |
| 4mb    | 42573.71| 2325640| 1459622.3| x     |
| 8mb    | 42573.71| 2325640| 1459638  | 1853650|
| 16mb   | 159860.5| 2325640| 1909024  | 1853635|
| 32mb   | 637451.8| 2323299.4| 1909028.7| 1853632|
| 64mb   | 1595133.| 1878496| 1909024  | 1853635|
| Average| 495518.6| 2235743| 1729267  | 1853638|

| Table 2. Average memory reclaimed by all garbage collectors in massively threaded applications |
|-------------------------------------------------|--------|-------|-------|-------|
|        | SR GC  | PR GC  | TR GC  | INC GC |
| 4mb    | 1083892.8| 1104038.4| 2246259.2| x     |
| 8mb    | 1518372.8| 988884.8 | 2770545.6| 3414844.8|
| 16mb   | 1162908.8| 976179.2 | 2770553.6| 4381296|
| 32mb   | 3253869  | 988884.8 | 2770553.6| 3853624|
| 64mb   | 3561980.8| 642529.6 | 2770545.6| 4381299.2|
| Average| 2116205  | 940103.4 | 2665692  | 4007766|

Fig 1: Performance of the Garbage collectors in Memory Collectors in Threaded Applications
Fig 2: Performance of the Garbage collectors in Memory Collectors in Massively Threaded Applications

collector. Results are depicted in the table 2 and figure 2. The average in all the stack sizes for massively threaded application reveals that the memory reclaimed by the Incremental collector 47.19%, 76.54% and 33.48% more than the serial, parallel and the generational collector respectively.

5. CONCLUSION
The issue of the memory reclamation has been remained as the most important aspect for the designers of the garbage collectors. The issues of memory reclamation becomes important the devices with limited amount of the memory. Number of Garbage collectors are introduced for the efficient reclamation of the memory and generational garbage collector is the latest collector introduced for the job but the results are not encouraging. Witwas Srisa-an et. al.[9] also studied generational garbage collector and found that Generational garbage collector can reduce pause times however, it does not improve the garbage collection efficiency in terms of memory reclamation. The efficiency of a generational scheme is often worst than a semi space copying collector. However, if the minor generation is configured to be larger than 50% of the heap size, the efficiency may be greater than copying.

REFERENCES