Design of Real Time Reference Counting Technique For a Real-time Garbage Collector

Sanjay Sahu

Sri Vaishnav Institute of Science and Technology
Rajiv Gandhi Proudyogiki Vishwavidyalaya, Bhopal
E-mail: er.sanjaysahu@gmail.com

***Abstract* – "Garbage collection is a technique to automatically reclaim unused memory areas for future use." Real time systems keep growing more & more complex and increasing the challenges of developers is the usages of safe high level language such as Java. We present a real time garbage collector for java. There is significant interest in applying real time garbage collection to system and we implemented one that is "Real Time Reference counting Technique". The methods are represented in the context of the real time specification for java and demonstrate that it can recycle dead objects in bounded time. Our aim is to find a different effective algorithm for garbage collection technique. This will work as a well under normal condition as well as real time condition too. We have tried to increase the percentage of application throughput, promptness and footprint.**

***Categories and subject Description* - Automatic Memory Management (Garbage Collector) and Special Purpose System (Real Time System)**

Keywords* – *Garbage Collection, Real-Time, Reference Counting, Algorithms, Measurement, Java

I. INTRODUCTION

Currently there is a number of garbage collection techniques available [1,6].In this thesis we are introducing a new garbage collection techniques. In languages without garbage collection (i.e. Pascal, C, etc.), the programmer must write bookkeeping code to keep track of heap-allocated cells, and free them explicitly when they are no longer needed. This can make programs significantly more complex. In languages with garbage collection, the programmer need not worry about the accounting of allocated cells; this makes programs simpler and more clear-cut. To be able to maintain full control of the runtime behavior of a system, it must be possible to predict the amount of resources (e.g. CPU time and memory) that is required for any (virtual) machine level instruction and for all runtime system work. Note that using such a system does not prevent writing an unpredictable application. An example is an application that waits for external events, e.g. input from a user. First, it is not always possible to know when the event occurs, and second the data passed with the event may be unknown. Thus, developer must still follow rules to handle such cases. Early implementations of new languages are typically designed to be easy to implement and prove correct.

To be more specific, garbage collection algorithms may be designed to interrupt the application for short time periods in the general case, but it need not be guaranteed that it will collect all garbage memory before the system runs out of memory. If the memory runs out, the system can be stopped to collect the remaining garbage memory. Such stop may take a second or two, but that does not matter to these systems. Unfortunately many such techniques are called real-time garbage collectors.

A large contribution to the overhead comes from the memory that is needed to allocate objects while the garbage collector collects garbage memory. Examples of other parts that need attention are thread support, synchronization, messaging, and some complex instructions. This is, however, out of scope for this thesis. This paper is organized as the main contributions of this work as follows:- Section 2 describes previous approaches to realtime garbage collection and Section 3 some of common problem of real time garbage collection encountered. Section 4 presents an overview of our garbage collector with Comparing & evaluating an efficient garbage collection algorithm. Section 5 describes the design of the garbage collector. Section 6 concludes the paper and provides an outlook on future work.

II. GARBAGE COLLECTION RELATED WORK

Garbage Collection was invented by John McCarthy[1] around 1959 to solve the problem of

manual memory management in LIST PROCESSING (LISP). Garbage Collection (GC) is detection and reclaiming of unused or inaccessible data structure. It is a form of automatic memory management, which reclaims garbage or memory used by objects that are no longer in use by application. Java runtime provides various types of garbage collection in java, which you can choose based upon your application's performance requirement.

Each garbage collector has been implemented to increase the throughput and reduces the garbage collection pause time. There are many techniques used for garbage collection. But here Marking Technique is discussed for Garbage Collection and it is divided into two phases

(1) Marking Phase: - Where all nodes in use are marked.

(2) Sweeping Phase: - All unmarked nodes are returned to the available Space list. This Second phase is unimportant when all nodes are of fixed size. Following figure 1 shows a node structure. When nodes are of variable size, it is desirable to compact the memory size, so that all free nodes form a contiguous block of memory (which is known as Memory Compaction).

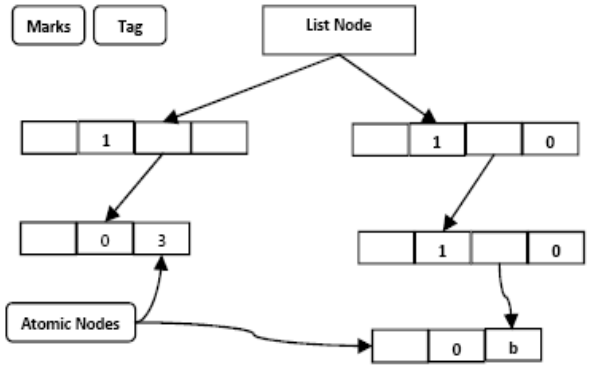

Fig.1 : The basic node structure

2.1 Marking

The marking can be defined as follows. There are some immediately accessible objects in the program; they are global and local variables, registers of the system. They are called root set. They contain references to objects in the memory. If a data structure is accessible each object referenced by one of the structure fields is accessible. The marking procedure can be defined recursively (consider, that the root set [2] consists of only one element): The procedure can be implemented without recursive procedure calls using a stack. The stack stores the references which should be examined. When a structure is marked, its fields that are references to non-marked structures are pushed into the stack. The marking procedure uses the reference on the

top of the stack which contains the root set at the start of the marking. The procedure runs until the stack is empty. It is a fast method but theoretically the size of the stack may be proportional to the size of the memory.

The main idea of this algorithm is that the edges of the directed graph (consisting of the structures and their references) can be directed reversely during the marking until leaves or already marked nodes are found. After that, the next node to be examined can be reached by going back on the reversed links which can be restored into the original directions this time.

2.2 Sweeping

This is the second phase in which, the whole data memory is scanned and the marked objects are collected in a contiguous area. The simplest method for reclaiming is that the unmarked cells are incorporated into one or more free lists. This may be accomplished by keeping a list for each object size commonly used in a program. These are called homogeneous free lists (H-lists). In addition, another free list, the M-list, contains cells of miscellaneous size ordered by their address. If possible, the reclaimed cells are linked into the corresponding H-list, otherwise, into the M-list. The memory allocation requests are satisfied from the H-list of the desired size, if it is not empty. Otherwise, a suitably cell is taken from the M-list, if possible. If it is larger than the desired size, it is split into two smaller ones and unrequired half is linked into one of the free lists. If the request cannot be satisfied, first a semi compaction is performed to move the cells of the H-lists to the M-list and bind together the adjacent free cells. If the semi compaction is still not enough, a real compaction is performed on the free cells

2.3 Compacting

In this two scans of the entire memory. The objective of the first scan is to perform the compaction and to build a "break table" which is used by the second scan to readjust the references. The break table contains the initial address of each "hole" - a sequence of unmarked cells and the size of the hole. The construction of the break table can be made without additional storage because it can be build up in the holes. It can be proved that the spaces available in the holes are sufficient to store the table. However, the table should be handled dynamically, rolling through the holes already filled with new data. At the end of the first scan, the live objects are collected into one end of the memory. The break table occupies the liberated part of the memory. The table is then sorted to speed up the

pointer readjustment done by the second scan. It consists of examining each pointer, consulting the table to compute the new position of the cell and changing the pointer accordingly. This algorithm is considerably slow because of the use of the holes and the binary search for each reference.

2.4 Reference Counting

Reference counting [3] is a well known Garbage Collection technique where each object contains a count of the number of reference to it held by other objects. If an object's reference count reaches zero, the object has become inaccessible, and it is put on a list of objects to be destroyed. In Computer Science, Reference Counting is a technique of storing the number of references, pointers, or handles to a resource such as an object or block of memory. It is typically used as a means of de-allocating objects which no longer referenced. In this a node is pointed by A & B and a one more node, hence its reference count is 3(Figure 2).

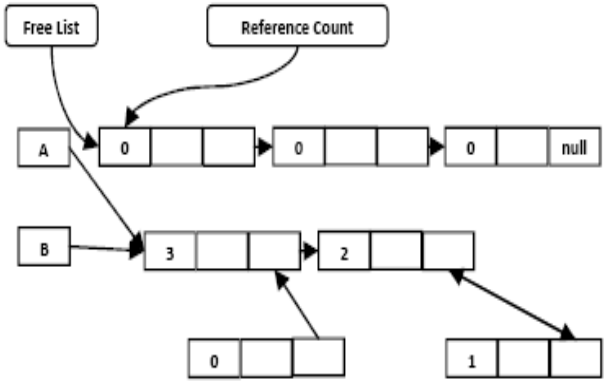

Fig. 2: Reference count

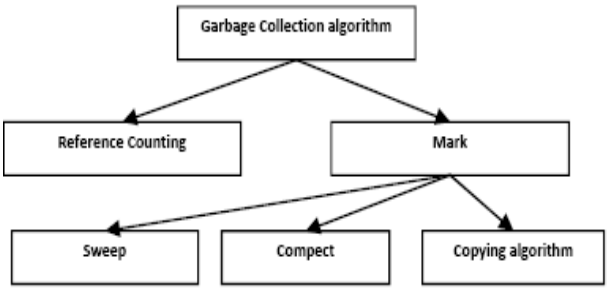

Fig. 3 : Classification of garbage collection technique

III. PROBLEMS WITH GARBAGE COLLECTOR

Previous approaches to real time garbage collection have generally suffered from a variety of problems. In this section we will describe these problems.

3.1 Fragmentation

A real time garbage collector must not avoid the real-time properties of the system as a whole. Obviously, it must not keep tasks from meeting their deadlines. However; it must also be possible to reason about the maximum consumption of a system. As fragmentation would make such reasoning difficult if not impossible, real-time garbage collectors must take it into account.

A solution to copy with fragmentation [4] is to entirely avoid external fragmentation. This can be achieved by organizing the heap in blocks of a fixed size. This eliminates external fragmentation at the expenses of a considerable amount of internal fragmentation.

Fragmentation can also be eliminated by defragmenting the heap, which requires the relocation of objects. During relocation, writes to the object that is about to be copied can potentially be lost or cause inconsistent data. In order toavoid consistency issues, objects are often copied automatically.

3.2 Sweeping Algorithm

The sweeping algorithm has three main disadvantages. First, the handling of the variable cells is cumbersome and fragments the memory. Second, the cost of the collection is proportional to the size of the entire memory because all live objects must be marked and all garbage's must be collected. The third problem is very interesting. This method does not move the objects - which can be seen as an advantage because of the unnecessary of pointer updating -therefore, the later allocated objects are interspersed with the earlier ones. This interleaving may be unsuitable for a system using virtual memory.

However, these problems are not always as bad as one could think. According to the statistics, the objects are often created in clusters and are typically active in the same time. The clever use of the mark bits can speed up the sweeping algorithms. The use of bitmap for mark bits makes possible the checking of 32 bits with one 32-bit integer operation. Since the objects tend to survive in clusters, this can greatly decrement the constant of the proportionality. The use of bitmap can also reduce the cost of the allocation by allowing fast allocation from contiguous unmarked areas rather than using free lists.

IV. OVERVIEW OF REAL TIME REFERENCE COUNT GARBAGE COLLECTOR

In Computer Science, Real-Time system [5] may be one where its application can be considered to be mission critical .Now let see the more detailed

classification of realtime systems. There are three types of systems discussed and are clearly distinguished by their features.

4.1 Interactive systems

They have an active dialog with their users. Normally the Users of an interactive system want response as soon as possible, but it is also important to keep the variance of the response limited. Examples of interactive systems are calculators, database front ends, editors, CAD-CAM (Computer Aided Design-Computer Aided Manufacture) systems, and data entry systems are all computer systems involving a high degree of human-computer interaction. Games and simulation are interactive systems. Web browsers and Integrated Development Environments (IDEs) are also examples of very complex interactive systems.etc.

4.2 A soft real-time system

It has specified deadlines, but an occasional slightly missed deadline does not lead to disaster. However, the quality of the result is reduced. Or in the other words it will tolerate such lateness, and may respond with decreased service quality (e.g., omitting frames while displaying a video). For e.g. - Multimedia systems, audio and video decoders, freezer are examples of soft real-time systems.

4.3 A hard real-time system

It has strict deadlines that should be guaranteed to be met at all times. Even an occasional slightly missed deadline in a hard real-time system could lead to a disaster. The anti-lock brakes on a car are a simple example of a real-time computing system — the real-time constraint in this system is the short time in which the brakes must be released to prevent the wheel from locking. Examples of hard real-time systems are airplane flight controllers and medical equipment's.

4.4 Generations

Most straightforward GC will just iterate over every object in the heap and determine if any other objects reference it. This gets really slow as the number of objects in the heap increase. GC's therefore make assumptions about how your application runs. Most common assumption is that an object is most likely to die shortly after it was created: called infant mortality. This assumes that an object that has been around for a while, will likely stay around for a while. GC organizes objects into generations (young, tenured, and perm) this is important. Ways to measure GC Performance

- Throughput- Percentage of time not spent in GC over a long period of time.
- Pauses - Application becomes unresponsive because of GC.
- Footprint- Overall memory a process takes to execute.
- Promptness- It is the time between object death, and time when memory becomes available.

If we were dealing with Real-time system then normal case of garbage collection described above will not work properly. So a new Concept of garbage collection is used that is Reference counting, which differs radically from other garbage collection techniques. This concept counts the number of references to every object and recycles the object if its reference count becomes zero. A reference counting memory handler is also used which consists of two main components that is – increment and decrement reference counters [7]. The decrement operation also handles the deallocation when reference counter becomes zero.

4.4 Decreasing Fragmentation

The garbage collection techniques such as the compacting ones are best in handling the fragmentation problem. By compacting memory, with each Garbage Collector cycle, fragmentation is eliminated. When memory is compacted, objects are moved from one memory region to another. Some of the garbage collection techniques do not move all the objects, while the others do.

We consider moving objects is too costly. Thus, an alternative technique should be searched. It can be done by using multiple free-lists; which contains objects of a specific size, and then the allocation can be done at constant time. When memory is divided into objects of certain sizes, fragmentation will not create a problem. If one free-list is to be used for every possible object size, many free-lists would have to be maintained.

The application would also need more memory, because as the different sized objects cannot reuse each other's memory. Using a fixed number of free-lists is a better option. Then each free-list holds memory regions of a specific size, e.g. powers of 2. Memory is allocated from the free-list containing the smallest memory regions which should be large enough to hold the object. The size of memory regions, the number of free-lists, and the number of regions in each list can be tuned at compile time. Therefore a large-object-region is to be used. The memory manager will handle these regions which should make system's execution-time predictable, fast and fragmentation safe. Compacting the large-object-region is not a good option, since large objects are the ones which cause more problems when memory

is compacted. Regions are safe form the problem of fragmentation which gets reduced because there are only large object residing currently in those regions.

In many real-time applications, currently static allocation is used.

4.5 Improving Performance

To allocate memory on the stack the allocation statement should only be executed a limited number of times per method activation and the objects should not be referenced by any method which is an ancestor in the call graph, i.e. there should not be a path from the referring method to the method which contains the allocation statement. Recent studies suggest that as many as 56% of the allocated objects could be allocated on the stack in some Java applications.

The study was done using run-time analysis, so our results may differ. The benefit of stack allocation is that these objects need no reference count. Using further analysis it is possible to find local references which only refer to stackallocated objects. These references need no special reference assignment. Thus, those references cause no overhead at all! Using stack allocation, the reference assignment routine becomes slightly more complicated. Instead of checking whether a reference is null or not, it must be checked whether the reference is to the stack or not.

When stack allocations have been added to the code, a peephole optimization technique proposed by Barth removes redundant reference count updates. Barth enumerates four cases where reference counts can be canceled.

1. The reference count can be set to one immediately, if an allocation is followed by an assignment.
2. If an object is allocated and the reference to it is immediately lost, code can be in lined to free the object.
3. Both the updates can be removed if a reference count is incremented and immediately decreased.
4. Both updates can be removed if a reference count is decremented and immediately incremented

According to tests by Barth, after allocations the first case eliminates almost all increments, while the other three cases are less common. Further tests have to be conducted to see whether this technique is worth using.

V. DESIGN OF RTRC GARBAGE COLLECTOR

By using features of java an application is developed which is based on RC and RTRC technique. Starting with RC, its working is described by process flow diagram (Figure 4). The RC of object reference is incremented and throughput is calculated. On the other hand RC is decremented and promptness is calculated by using gc() method.

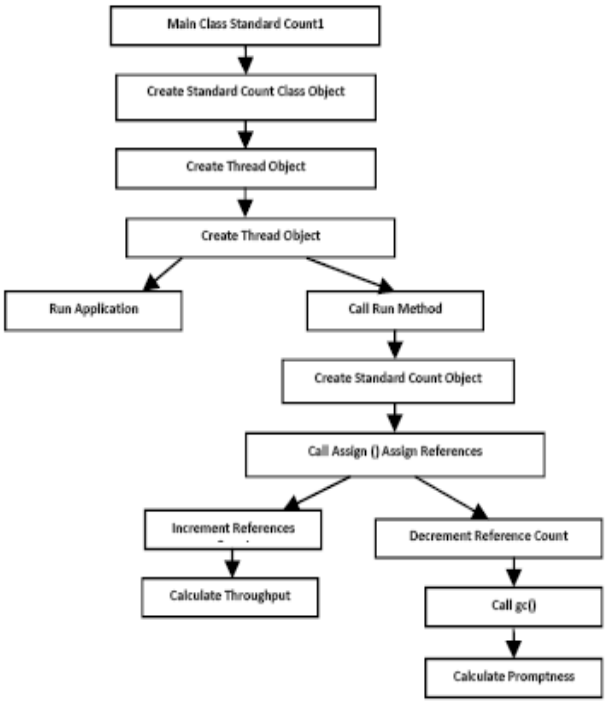

Fig. 4: Shows process flow diagram of RC technique

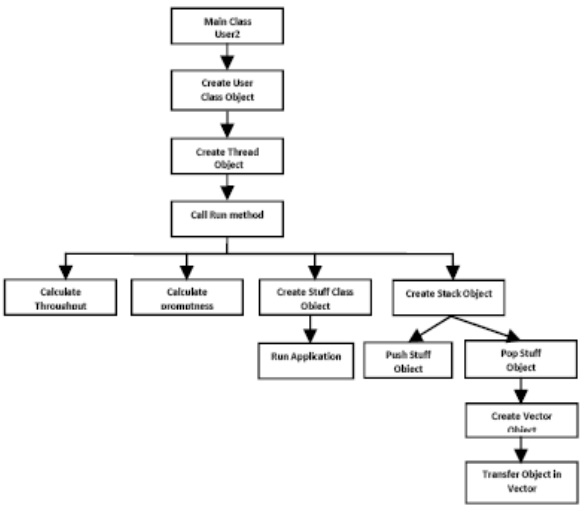

Fig.5: Shows process flow diagram of RTRC Technique

Next is RTRC (Figure 5), in this method our application is executed inside the constructor of stuff class. The object of class stack is also created in run ( ) method. Every stuff object is pushed inside the stack

.When the object in class stuff is no longer needed, it is popped from stack .Instead of sending the object for GC it is transferred in a vector also known as free list.

Simple reference counts require frequent updates. Whenever a reference is destroyed or overwritten, the reference count of the object it references is decremented, and whenever one is created or copied, the reference count of the object it references is incremented.

The Application has the following features:-

1. The application output of performance of both the algorithms is shown by using a simple bar graph indicating the performance of applications in terms of overall throughput and promptness.
2. Both the algorithms have been implemented in multithreaded environment.

It is clearly shows the standard count algorithm decreases the overall throughput as promptness of application as it uses the mode of simple reference counting.

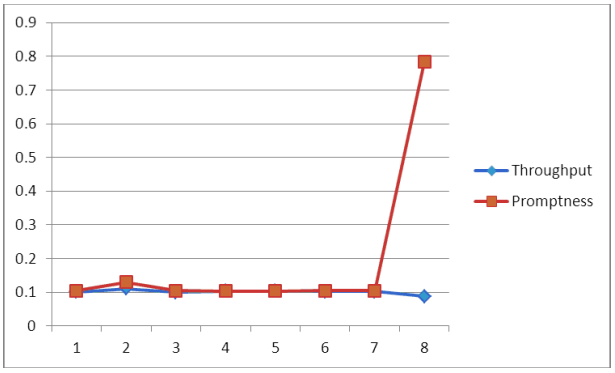

Fig. 6 : Output using RC technique (an application is running concurrently)

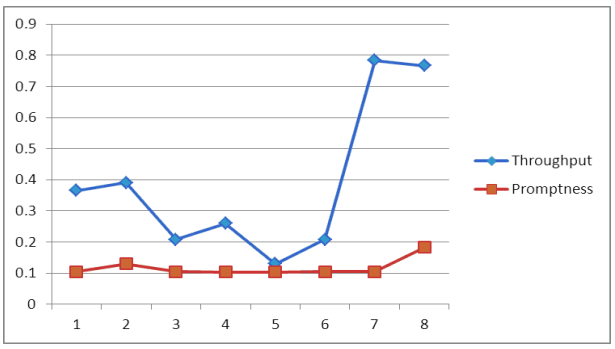

Fig. 7 : Output Using RTRC technique (an application is running concurrently)

It is clearly shows the RT algorithm increases the overall throughput of application as instead of destroying an object as soon as its reference count becomes zero, it is added to a list of unreferenced objects, and periodically (or as needed) one or more items from this list are destroyed.

By following the process flow diagram of RC & RTRC, an application is developed during implementation which is based on RC & RTRC technique. For an ideal case according to meaning of Throughput & Promptness, a system should be efficient when its throughput is more & promptness is less .Now proceeding towards the result and evaluation, which is done by saving the values of throughput & promptness obtained from application. During saving the values, application is concurrently running with other applications at back which provides a real-time effect and then the real-time values of throughput & promptness are saved. Hence RTRC is a good technique because throughput is increased and promptness is decreased, which is a desired result.

VI. CONCLUSION

In this work it seems that the promptness of the system is decreased and the throughput is increased when RT reference counting is used as compared to the standard reference counting.

In this thesis the two garbage collection techniques are our point of attention, which will help or gives us a prediction about the behavior of the system when used.

As already stated that- what garbage collection is, which a back end work done by operating system. This garbage collection is very important aspect of memory management. This garbage collection is carried out when system is in need of resource that is memory, then operating system initiates garbage collection work at the back end. In this report a Marking Technique for garbage collection is discussed, which seems not suitable to work in a Real Time Environment. After that a new Reference Counting concept was introduced which seems good but no promising enough to work in Real-Time. Real-Time Reference Counting is seems good for carrying Garbage Collection which should be more deeply analyzed prior to implementation.

VII. REFERENCES

[1] John McCarthy. Recursive functions of symbolic expressions and their computations by machine, part I. Communications of the ACM, 3(4):184-195, April 1960.

[2] W. Puffitsch and M. Schoeberl. Non-blocking root scanning for realtime garbage collection. In Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded

Systems (JTRES), pages 68–76, Santa Clara, California, Sept. 2008. ACM Press.

[3] M. Schoeberl and W. Puffitsch. Nonblocking real-time garbage collection. ACM Trans. Embed. Comput. Syst., 10:6:1–6:28, August 2010.

[4] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, pages 146–159, Toronto, Canada, June 2010. ACM Press.

[5] F. Siebert. Concurrent, parallel, real-time garbage-collection. In J. Vitek and D. Lea, editors, Proceedings of the Ninth International Symposium on Memory Management, pages 11–20, Toronto, Canada, June 2010. ACM Press.

[6] P. R. Wilson: Uniprocessor Garbage Collection Techniques. Proc. Of the 1992 Intl. Workshop On Memory Management. Springer-Verlag Lecture Notes in Computer Science series.

[7] Deutsch, L.P., and Bobrow, D.G. An Efficient, Incremental, Automatic Garbage Collector. Commun. ACM 19, 9 (September 1976) 522-526.

◈◈◈