International Journal of Computer Architecture and Mobility
(ISSN 2319-9229) Volume 1-Issue 4,February 2013

Implementation of a Real-time Garbage Collector Technique

Sanjay Sahu, Anand Rajavat
Sri Vaishnav Institute of Science and Technology
Rajiv Gandhi Proudyogiki Vishwavidyalaya, Bhopal
er.sanjaysahu@gmail.com

Abstract- "Garbage collection is a well known technique to automatically reclaim unused memory areas for future use." Almost all such methods require large amounts of additional memory. Conventional garbage collection techniques are however not suited for use in real-time systems. There is significant interest in applying real time garbage collection to system and we implemented one that is "Real Time Reference counting Technique" that increases memory efficiency by about 50 % compared to the most memory efficient previously presented predictable garbage collector. 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.

Key words-Garbage Collection, Real-Time, Reference Counting, Algorithms, Measurement, Java.

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

I. INTRODUCTION

Currently there is a number of garbage collection techniques available [1,6].In this thesis presents work in the area of automatic memory management for real-time systems. The motivation of the thesis is to be able to develop real-time systems using modern languages such as Java. Since these languages commonly use automatic memory management or garbage collection (GC), which traditionally has had an unpredictable runtime behavior, we could either try to eliminate the need for GC using manual techniques, or we could develop GC techniques for these systems. Since GC is such a powerful tool to eliminate memory related programming errors, we decided to develop techniques to use GC in real-time systems. During this work three other GC techniques for these systems have been published. The main advantage of our work compared to the other three is that memory utilization efficiency increased by about 50 %. We have also developed an optimization for incremental garbage collectors and a static garbage collector that aims to eliminate the need for runtime garbage collection.

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.

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.

2.1 Mark-and-sweep

Mark-and-sweep collectors perform the garbage collection in two phases.

(1) Marking Phase: - Where all nodes in use are marked. Mark-and-sweep collectors perform First, live memory is marked by traversing the object graph starting at the roots. Next all unmarked memory is reclaimed in the sweep phase. The algorithm starts by marking the roots. Marking an object includes finding its children and marking them. By marking the roots, all reachable objects will be marked. The sweeping phase traverses the heap and all unmarked objects found are reclaimed.

It is a common choice to start the garbage collector from the memory allocation function. In a non-incremental algorithm the collector is often started when the system runs out of memory. When using incremental collectors, some work is commonly performed every time memory is allocated. The amount of work in each increment is often proportional to the amount of allocated memory.

(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).

2.2 Compacting

Compacting the heap includes moving live regions and updating pointers to the regions which have been moved. 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.

2.3 Copying Algorithms

A copying garbage collector uses a heap which is divided into two or more sub-heaps. This section describes two sub-heap versions. The two sub-heaps are labeled to-space and from-space, respectively. All objects are allocated in to-space where all live memory regions reside. When to-space is full, a flip is performed. First the labels are swapped, i.e. to-space becomes from-space and from-space becomes to-space. Next, the roots are copied from from-space (previously called to-space) into to-space. When an object is copied, all children of that object are copied too. When all live objects have been copied, all pointers have to be updated to point to the new copies of the objects. Finally the garbage collector hands over control to the mutator. An advantage of a copying garbage collector is that when the objects are copied, they are compacted. Thus, a copying garbage collector does not suffer from external fragmentation. Because the memory is compacted and placed at one end of the heap, allocation of n bytes can be done by simply sliding a pointer n positions in the free memory region. An advantage of the technique is that the running time of the garbage collector is proportional to the number of live objects. Thus, a large heap size does not affect the running time of the collector, and the collector can be run less frequently.

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 1).

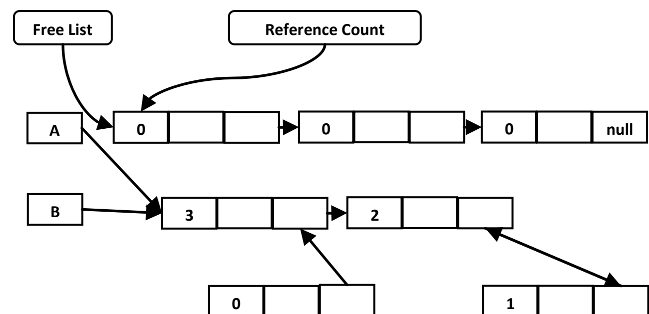

Fig.1 Reference count

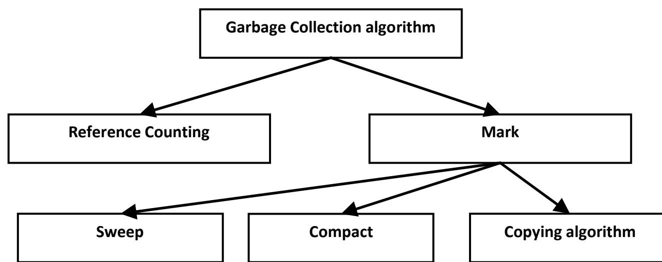

Fig. 2 Classification of garbage collection technique

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 Recursive freeing

When the last reference to a data structure is deleted, all objects in that structure are reclaimed. If the data structure is large, in the worst case all objects on the heap, this behavior causes long interrupts in the execution of the system. Since recursive freeing can occur anywhere when reference counts are decremented, e.g. at assignments, the WCET becomes very pessimistic. To be able to use reference counting in hard real-time systems, recursive freeing must be eliminated.

3.2 External fragmentation

External fragmentation occurs when small regions of free memory exist between the allocated objects. Even though there is enough memory to allocate a new object, there may be no contiguous region that is large enough. Thus, the allocation fails even if there is memory available. External fragmentation rarely causes problems, since clever allocation strategies keep it small. Even if the heap becomes more fragmented, most systems have enough memory to continue. But to predict the external fragmentation in advance is a difficult problem with no known solution. In a hard real-time system, the worst case memory usage must be known in advance, thus external fragmentation must be handled.

3.3 Worst Case of Execution Time

WCET of allocation Allocators usually have a WCET that is proportional to the heap size or to the logarithm of the heap size. The average execution time is normally much shorter due to pools of blocks of common sizes, but the average execution time is of little use in hard real-time. Thus, execution time should be improved for the technique to be competitive.

3.4 Inability to reclaim cyclic garbage

Since the internal references of cyclic data structures keep all reference counts above zero, the objects that are part of cycles cannot be reclaimed. Many systems can be implemented to have no dead cyclic data structures. However, for the technique to be useful when dead cycles cannot be avoided, it must be possible to reclaim them.

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 real-time systems. There are three types of systems discussed and are clearly distinguished by their features.

4.1 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.

4.2 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.

4.3 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 de-allocation 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.

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 stack-allocated 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

In this section, the design of a RTRC is presented. This design does not cover recovering cyclic data structures, since any of the techniques de-scribed above can be used. As stated above, many systems can quite easily be designed not to produce any cyclic garbage, especially hard real-time systems where the developers must have full control of the execution of the system.

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 3). 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.

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.

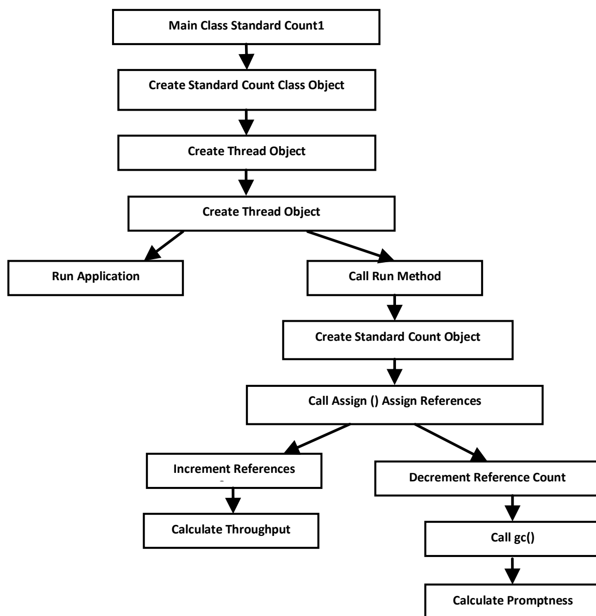

Figure 3 Shows process flow diagram of RC technique

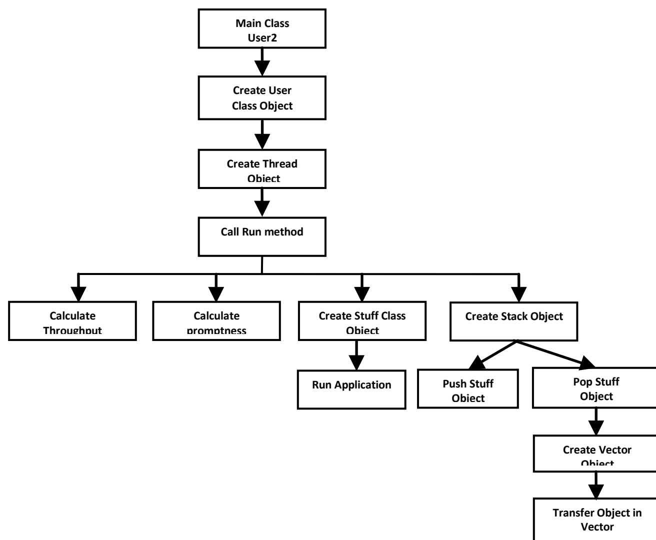

Figure 4 Shows process flow diagram of RTRC Technique

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.

5.1 The allocator of RT-Reference Counting

```
Reference new (int size)
   Reference ref;
   if inBackroundProcess()
        deallocate(); // Algorithm 4.3
        end  // The size is adjusted fit in the free-list or in
the large object region
        size = adjust(size);
        switch size
        case 2: ref = fl2; fl2 = fl2.next; break;
        case 4: ref = fl4; fl4 = fl4.next; break;
        case 8: ref = fl8; fl8 = fl8.next; break;
        ...
        // Allocate from the large object area
        default: ref = allocate(size); break;
        end // Call constructor
        ref.init();
        return ref;
        end
```

5.2 Write-barrier using RT-Reference Counting

```
release(Reference ref)
ref.refCount = ref.refCount - 1;
if ref.refCount == 0
// The type field refers to type specific data
// The tbfList is the to-be-free list of
// of the type
synchronized ref.type.tbfList
ref.next = ref.type.tbfList;
ref.type.tbfList = ref;
end
end
end
assignReference(Reference lhs, Reference rhs)
if rhs.onHeap()
rhs.refCount = rhs.refCount + 1;
end
if lhs.onHeap()
release(lhs);
end
lhs = rhs;
end
```

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

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.

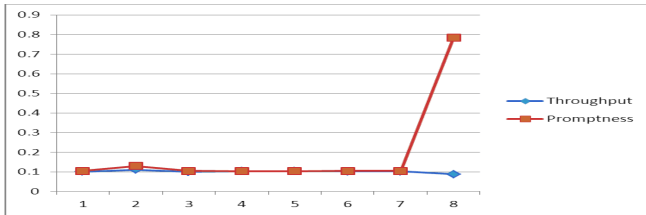

Figure 5 Output using RC technique (an application is running concurrently)

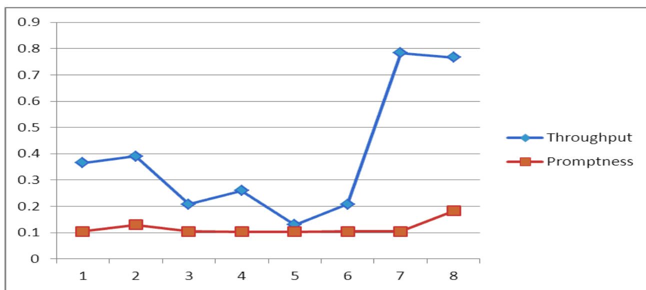

Figure 6 Output Using RTRC technique (an application is running concurrently)

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.

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 real-time 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.