International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 4 Issue: 11 143 - 150

Conservative Multi-Generational Age-Based Garbage Collection with Fast Allocation

Shailesh Arrawatia
Department of Computer Science & Engineering
APEX Institute of Engineering & Technology, Jaipur
Shailesh24790@gmail.com

***Abstract*—** In the era of today's technology, Garbage Collectors have high mortality and high efficiency because they look and remove garbage memory blocks among newly created objects. Many very newly created objects are included into these objects which are still live and easily can be identified as live objects. Generational Garbage Collection is a technique which is based on newer objects where the older objects are pointed by these newly created objects; because of this, these type of algorithms earn more efficiency than other garbage collectors. The only one way called "Store Operation" is used to a formerly created objects for pointing to a newly created objects and many languages have limitations for these operations. Recently allocated objects are focused more by a Garbage Collector and these objects can give more support to the above mentioned issue. The efficiency of such type of Garbage Collectors can be measured on the basis of allocation and expenditure type than the disposal of objects. In this paper, we have studied various techniques based on Generational Garbage Collection to observe object structures for producing better layout for finding live objects, in which objects with high temporal weakness are placed next to each other, so that they are likely to locate in the same generation block. This paper presents a low-overhead version of a new Garbage Collection technique, called Conservative multi-generational age-based algorithm which is simple and more efficient with fast allocation, suitable to implement for many object oriented languages. Conservative multi-generational age-based algorithm is compatible with high performance for the many managed object oriented languages.

Keywords*—Garbage Collection, Dynamic Memory Allocation, Conservative.*

*****

I. INTRODUCTION

Garbage collection (GC) involves pass through the data objects which are live during the execution of a program and this process is just like parallel even in subsequent programs. Generational garbage collection [1] is an efficient technique for finding and reclaiming of unreachable heap data objects that are required by user for reusing the heap space. Reclaiming of short-lived objects are done very quickly and efficiently while another long-lived heap objects are adjusted in the regions of the heap. These long-lived heap objects are subject to more relatively uncommon collections. It can manage a wide range of heap spaces with generally short pause times, these predominantly affecting the time of collection to perform short term collections. Eventually, however, the region(s) containing old data objects will be adjusted by filling copied objects and this behavior necessary for doing major collection. Typically, this operation for collecting major data object are more expensive because the earlier procreations are much larger than the new one. Furthermore, the collection of old generation needs accurate collection of all younger data objects procreations so, regardless of the actual number of procreations, the entire heap will eventually require collection. The program execution activity [2] and garbage collector are interleaved in the above way of collection. In implementing this "barrier less" scheme, people can easily change the complete behavior of copied data objects at the time of garbage collection. The object land can be attempted to enter in the self-scavenging code during this attempt. At the time compiling, another new alternative approach is to be introduced for specializing the entry code for each collected data objects. These eradicate the need for the extra word as we can simply turn from one data objects to the other copied data objects at the time of garbage collection. This paper focuses on the complete detail of the proposed garbage collection technique and shows how this code specialization can be made to work in practice and the effect of reclaiming the dynamic heap space can be evaluated more efficiently upward for garbage collector.

A. Background

The Baker's incremental collection algorithm [2] is usual for readers at the current time while many people have considered that the reader is usual easily with the fundamentals of generational garbage collection [4]. In this paper, it is assumed that complete collection of garbage data objects are performed by copying live objects from one space to another space. The copying of live data objects is synonymous in Baker's algorithm with fast allocation and evacuation. These evacuated data objects are called scavenged. The old generation and the young generation are two procreations to follow the collection in the generational garbage collection although this proposed method of garbage collection can be settled to adjust the procreations number of procreations are arbitrary in nature. The assumptions are taken in this paper as many data objects are age-based in nature during the copy of data objects from one generation to another generation.

B. Motivation

We have taken an objective for performing the garbage collection in parallel with help of collector which perform collection in shared-memory that is employing to achieve faster and efficient reclaiming of data objects with compared to one processor that could do alone. In this paper, a collector is presented for better reclaiming of unreachable data objects and this collector is based on different generations where copying is done from one generation to another generation by copying

collector. Many procreations are created by dividing the heap as some younger procreations and generation n is collected for n-generational garbage collector. We have implemented this algorithm for 3 generations and it can be improved for n generations. In this proposed method, the complete set list of all the heap data objects is pointed by younger generation while many implementation lists the all the data objects.

This paper is arranged as Section 2 gives complete review of the related literature survey on various garbage collector algorithms, Section 3 of the paper gives the proposed approach for conservative multi-generational age-based Garbage Collection to enhance allocation. Section 4 presents our proposed algorithm. Finally, Section 5 describes performance matrices and Section 6 shows conclusion and future work for the proposed algorithm.

II. LITERATURE SURVEY

The two fundamental approaches are implemented in the decade of 1960 and these approaches provide complete storage redemption. The name of these approaches are declared by researchers as namely tracing [5] and reference counting [6]. So this was a great achievement of work done in the field of garbage collection since that time and according to time, numerous advancement has been developed in both the approaches. Some of the major important advancements are done constantly in copying collection [4] which is based on generational collection. Many more approaches are implemented for reclaiming the unreachable data objects and named as soft real-time collection [2], hard real-time collection [3], Mark and Copy [7], Space Tracing Collection[5], Assigned Garbage Collection. Some of the major advances have been proposed for incremental loosening [4] in the collection of reference counting. Another garbage collection technique based on deferred reference counting [2] and compile-time reclaiming of heap objects is also more efficient for reclaiming the younger data objects with multiprocessor parallel collection [1].

Reference counting collectors identify unreachable objects and reallocate them as soon as much fast counting and these unreachable objects are no longer reachable referenced [8]. The association of the reachable data objects with each object have a reference count that can be incremented during the garbage collection each time and a new pointers to the data objects are created and decremented each time one is destroyed. If reference count falls to zero, the reference counts for immediate descendants are decremented and the object is reallocated. Unfortunately, reference counting collectors are expensive because the counts must be maintained and it is difficult to reclaim circular data structures using only local reach ability information.

The best reference counting collectors have very low and uniform latency impact on an application as demonstrated by the Ulterior Reference Counting [10] collector. However, they have historically suffered from lower throughput compared to tracing collectors. The work of Shahriyar et al. [56, 57] has made reference counting collectors competitive, but does so by incorporating background tasks and pauses. Unfortunately Shahriyar doesn't report the latency impact of these changes.

Mark sweep collectors [9] are able to reclaim circular structures by determining information about global reach ability. Periodically, when a memory threshold is exhausted the collector marks all reachable objects and then reclaims the space used by the unmarked ones. Mark sweep collectors are also expensive because every dynamically allocated object must be visited the live ones during the mark phase.

Copying collectors [10] provide a partial solution to this problem. These algorithms mark objects by copying them to a separate contiguous area of primary memory. Once all the reachable objects have been copied the entire address space consumed by the remaining unreachable objects is reclaimed at once garbage objects need not be swept individually. Because in most cases the ratio of live to dead objects tends to be small by selecting an appropriate collection interval the cost of copying live objects is more than o set by the drastically reduced cost of reclaiming the dead ones. As an additional benefit spatial locality is improved as the copying phase compacts all the live objects. Finally, allocation of new objects from the contiguous free space becomes extremely inexpensive. A pointer to the beginning of the free space is maintained allocation consists of returning the pointer and incrementing it by the size of the allocated object. For best performance a collector should minimize the number of times each reachable object is traced during its lifetime Generational collectors deed the experimental observation that old objects are less likely to die than young ones by tracing old objects less frequently. Since most of the dead objects will be young only a small fraction of the reclaimable space will remain unreclaimed after each collection and the cost of frequently retracing all the old objects is saved.

Generational collectors [11] have been implemented successfully in prototyping languages such as LISP, Modula, Smalltalk, etc. These languages share the characteristic that pointers to objects are readily identifiable or hardware tags are used to identify pointers. When pointers cannot be identified, copying collectors cannot be used for when an object is copied all pointers referring to it must be changed to react its new address. If a pointer cannot be distinguished from other data then its value cannot be up dated because doing so may alter the value of a variable.

Conservative collectors [1] may be used in language systems where pointers cannot be reliably identified. This class of collectors makes use of the surprising fact that values that look like pointers ambiguous pointers usually are pointers. Misidentified pointers result in some objects being treated as live when in fact they are garbage. Although some applications can exhibit severe leakage.

III. PROBLEM STATEMENT

A. Gap analysis among Garbage Collectors

Most of the garbage collectors have a performance gap due to large heap size and slow speed of copying phase. This is because of the combination of the factors like slow allocation of heap sequence and the periodic copying operation when the new empty object block is required for the free block sequence of particular size according to the locality of the resulting new heap blocks. We have removed above discussed problem in this proposed model of garbage collector by using child-parent counting model. In this child-parent counting, we have applied type of mark bit of the heap blocks of different size classes like small, medium and large. So these different sized blocks are used to copy live blocks according to their size.

Another fundamental issue of the garbage collectors is that in small heaps, there is an internal and external fragmentation problem of memory heap blocks. This issue is resolved in this proposed developed model of garbage collector by providing different class size of memory heap blocks. The dynamic compression of live object blocks are highly used to reduce the heap memory requirements of the running application. Mostly, people are used two strategies: first, the heap space is exhausted and another one is to perform the operation of compression on objects what are infrequently accessed objects. Also, people avoid the allocating space of used data objects.

So above discussed issues has been resolved in this proposed model of conservative multi-generational garbage collector and the detailed model of proposed work is discussed in subsequent sections.

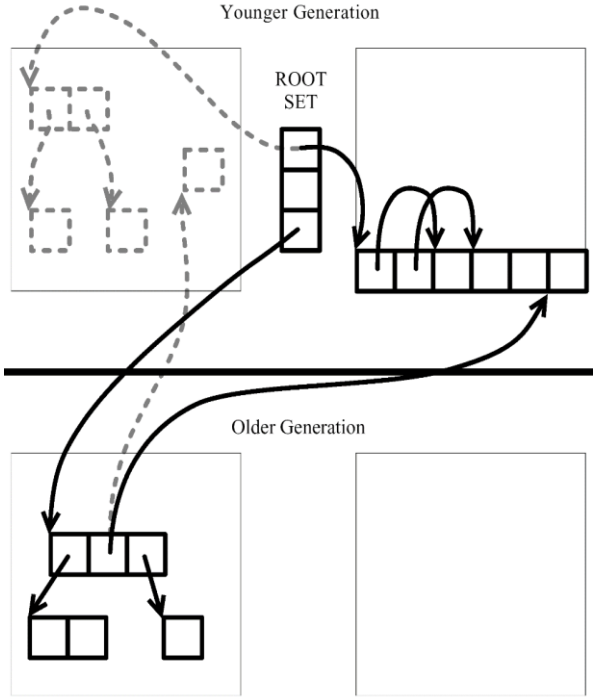

Figure 1. Generational Collector after collection

B. *Background of Proposed Work*

According to many researches, most of the object live for a short period of time, some live for medium amount of time and very lower number of objects live for a longer period of time [6]. So in this the heap is divided according to the age of data objects and garbage is collected according to age of data objects. So this way the useless copying of short life objects has not been taken place.

Promotion Policies [4]: In this, it is decided that when one object will promoted to next generation .In this it is seen that we have to take into consideration early betterment and late betterments. For long lived object early promotion is better than late promotion. For short lived object early promotion is better than live object.

Intergeneration Reference [4]: In this we must be aware that any pointer variable from older generation refers to younger generation. As doing garbage collector in younger generation we must trace the pointers reference from older to younger generation. As some objects are unreachable from other references except this one. So we have to trace these references and for this we have number of techniques to implement.

For the advancement policy [6] we have taken gc_count variable whose value will decide In which generation the data objects should be place .We have divided our heap into two generation and in one generation we have taken two buckets so gc_count will decide that weather we have to transfer data object from one bucket to another or from one generation to another. Now it is necessary to trace inter-generational link as it is possibly that garbage collector will collect the data object from younger generation in spite of the fact that it has been pointed by the data object in older generation. So for this we have taken remember set so that we trace at each store operation the link of data objects and if it being a pointer from older generation to younger generation than make that entry in remember set (hash table) and during each copying phase remember set is scan so that if it is entry in remember set than that memory location or data object being transfer from one younger generation to old one.

C. *Proposed Approach*

As in garbage collector we have to keep trace of different information regarding the data objects being allocated by dynamic memory allocation functions. So for this we have built our own memory allocation function which is use to keep the record the information about the objects which is being allocated and reallocated. Now for doing garbage collection we must maintain the information of the data objects so that we can operate on them and get the appropriate information's to mark and the objects. So we make the structure of the data objects as following:

```
struct block_structure
{
void * object_pointer;
char mark_copy_flag;
struct block_structure* next_block;
};
//object_pointer: poniter to allocated block
//mark_copy_flag: used for storing the information
//which is used during mark and copy phase.
//next_block: next pointer of this list.
```

Bit Position of Mark Copy Flag

BP7	BP6	BP5	BP4	BP3	BP2	BP1	BP0

Details of individual bits:

B0: Check Bit- During mark phase it is used for interleaving objects. If this bit is 0 than corresponding object has to be processed for interleaving. If it is 1 than object has already been processed. And during copy phase it denotes whether object has been copied or not. After marking it will be 1 if object is live and after copying it will be 0.

B1: mark bit- It denotes whether object is live from stack or not. After marking it will be 1 if object is live from stack and after copying it will be 0

B2: child bit- It is used to denote whether the corresponding object has a child or not. If it is 1 than object points to some other objects.

B3: parent bit- It denotes whether object is live from any other live objects or not. After marking it will be 1 if object is live and after copying it will be 0

B4 and B7: unused.

B5 and B6: count bit- It is used to check in which generation the object will be copied. Initially it will be 0 after allocating new objects. If object is copied to another generation it will be incremented. "00" stands for object is in bucket 1 of generation one, "01" stands for object is in bucket 2 of generation one, and "10" stands for object is in generation two.

above shown the structure of data object in which we have pointer to next block , size of data object and in 8 bit we have all the variables which is used to mark the data objects and copying the data object and one bit is for maintaining the remember set value to trace inter-generational link.

D. Design Goal of a Proposed Approach

Our main goal is to improve the efficiency of reclaiming the unused memory which is unreachable from live objects. Many people believed that a complete garbage collection would be made it possible to find live objects easily in generational garbage collection. Our final design goal is to make a design architecture that can support to find optimum live objects to provide a better garbage collection.

IV. PROPOSED ALGORITHM

We have design the algorithm for memory allocation function and free function as shown below:

A. Algorithm for GGC_malloc ()

i. Select a block and repeat until whole free list is scanned
ii. If(block>=size) than
{
(a) If (block>size)
{
Break the block as a tail part is equal to size + sizeof (Header)
Set the size of block at the starting of newly block
Update the (size) field in the remaining block
}
(b) If (block==size) Remove the block from free list.
(c) Set the entry of this newly allocated block in object maintaining list by using GGC_maintain_object_list (pointer to allocated block)
(d) Return the pointer of this newly block to user
}
iii. If (block is not found) than Add extra free space to free list using allocate_extra_memory_to_gen () and repeat above process once again.
iv. If(constraints true) call GGC_garbage_collector()

B. Algorithm for GC_free ()

i. Find the entry of this block into object maintaining list.
ii. Remove this entry from the object maintaining list.
iii. Add this block into the particular free list.

C. Algorithm for GGC_garbage_collector()

In this phase we have to mark the data object which is being lived from the stack as well as the object resides in the heap. This is one of the important phase in any garbage collector technique because this give the way to which distinguish between live and dead objects. When the garbage collector is called the three operations will be performed and these are:-

i. Mark()
ii. Copy()
iii. write barrier()

Algorithm for mark()

i. find stack high pointer and current pointer
ii. lower_address =current pointer, upper_ address =high pointer.
iii. While(lower_address < upper_address)
a) Get address which is stored in stack at current pointer.
b) If this address points to any block in heap than set mark bit 1 for this block in object maintaining list.
c) Increment lower_address.
iv. Select object while(mark=1 and check=0)
a) Set check=1 for this object.
b) Set lower_address =start address of block, upper_ address = end address of block.
c) Repeat step 3.
d) If any object referenced from this block than set child bit=1 for this block and set parent bit for referenced block.

Now for copying we have taken the concept of cheney's algorithm [9] in which we have two pointers variables in which one is used to trace the copying of data object and other used to trace the pointer adjustment.

Copy phase: In this, the copy the reachable data objects from one heap location to another heap location. Copy of the data objects is done and we update the stack address of the corresponding objects. In this we also perform copying of interleave data objects ad update their entry in the corresponding objects.

Algorithm for copy ()

1) start=end=initial address of To Space
2) select a root block until all roots are traversed.
3) set mark=0
4) copy this block to next generation according to GC_count().
5) Update new address of this block in stack and set check=0.
6) end=end+size of this block
7) while (start<end)
a) If (child==1) find all referenced block from this block and copy them to To_Space. And set child=0.
b) Update all parent addresses of referenced block. Set check =0 for referenced block.
c) end=end+size of referenced block.
d) start=start + size of this current parent block.
8) goto step 2.

Write barrier: This is important phase as in this we trace the data objects which have the link from older generation to younger procreations and then scavenged that particular objects, as this intergeneration link can cause a problem of write barrier so that it is needed to keep trace this link and scavenged them.

Now for write barrier we have to trace at every store operation the pointer from older generation to younger generation and put its entry into the remember set .When the

copying is done than data object been trace for the link and if it is available u can scavenged it into the older generation.

Algorithm for Write_barrier ()

i. Trace object for each store operation.
ii. If object pointed from older generation to younger generation than put the object address in remember set.
iii. During copy trace the remember set if object found in remember set than scavenged the object into older generation.

V. PERFORMANCE MATRICS

In testing, we have taken number of various matrices to calculate the efficiency of proposed Garbage Collector and shows the relationship with in themselves so that that we can analyze our proposed solution.

We have tested these performance matrices by using the various test benches which are shown in the subsequent sub-section.

A. Testbenches

a)

```
#include"ggc.h"
//test bench
void main()
{
int x,i;
stack_high_ptr=&x;
block rtemp;
void print_list(struct link *c);
struct link *head,*tempxyz,*a ,*b,*c,*d,*e;
size_t size_total=0;
a=(struct link *)GGC_malloc(sizeof(struct link));
a->num=10;
b=(struct link *)GGC_malloc(sizeof(struct link));
b->num=20;
c=(struct link *)GGC_malloc(sizeof(struct link));
c->num=30;
d=(struct link *)GGC_malloc(sizeof(struct link));
d->num=40;
e=(struct link *)GGC_malloc(sizeof(struct link));
e->num=50;
GGC_garbage_collector();
}
void print_list(struct link *c)
{
while(c!=NULL)
{
printf("\t%0x",c);
printf("\t%d",c->num);
c=c->next;
}
}
```

b)

```
#include"ggc.h"
//test bench
void main()
{
int x,i;
stack_high_ptr=&x;
block rtemp;
void print_list(struct link *c);
struct link *head,*tempxyz,*a ,*b,*c;
size_t size_total=0;
a=(struct link *)GGC_malloc(sizeof(struct link));
a->num=0;
c=a;
for(i=1;i<50;i++)
{
c->next=(struct link *)GGC_malloc(sizeof(struct link));
c->next->num=10*i;
c=c->next;
}
c=NULL;
printf("\n&a=%0x",&a);
print_list(a);
GGC_garbage_collector();
printf("\nafter collection\n");
print_list(a);
}
void print_list(struct link *c)
{
while(c!=NULL)
{
printf("\t%0x",c);
printf("\t%d",c->num);
c=c->next;
}
}
```

c)

```
#include"ggc.h"
//test bench
void main()
{
int x,i;
stack_high_ptr=&x;
block rtemp;
void print_list(struct link *c);
struct link *head,*tempxyz,*a ,*b,*c;
size_t size_total=0;
a=(struct link *)GGC_malloc(sizeof(struct link));
a->num=0;
b=(struct link *)GGC_malloc(sizeof(struct link));
b->num=0;
c=a;
for(i=1;i<10;i++)
{
c->next=(struct link *)GGC_malloc(sizeof(struct link));
c->next->num=10*i;
c=c->next;
}
c=NULL;
c=b;
for(i=1;i<10;i++)
{
c->next=(struct link *)GGC_malloc(sizeof(struct link));
c->next->num=10*i;
c=c->next;
}
c=NULL;
GGC_garbage_collector();
```

 ISSN: 2321-8169
143 - 150

```
}
void print_list(struct link *c)
{
while(c!=NULL)
{
printf("\t%0x",c);
printf("\t%d",c->num);
c=c->next;
}
}
```

d)

```
#include"ggc.h"
//test bench
void main()
{
int x,i;
stack_high_ptr=&x;
block rtemp;
void print_list(struct link *c);
struct link *head,*tempxyz,*a ,*b,*c;
size_t size_total=0;
a=(struct link *)GGC_malloc(sizeof(struct link));
a->num=0;
c=a;
for(i=1;i<10;i++)
{
c->next=(struct link *)GGC_malloc(sizeof(struct link));
c->next->num=10*i;
c=c->next;
}
c=NULL;
printf("\n&a=%0x",&a);
print_list(a);
c=a;
while(c!=NULL)
{
if(c->num==50)break;
c=c->next;
}
c->next=NULL;
GGC_garbage_collector();
printf("\nAfter collection\n");
print_list(a);
GGC_garbage_collector();
printf("\nAgain after collection\n");
print_list(a);
GGC_garbage_collector();
printf("\nAgain after collection\n");
print_list(a);
GGC_garbage_collector();
printf("\nAgain after collection\n");
print_list(a);
}
void print_list(struct link *c)
{
while(c!=NULL)
{
printf("\t%0x",c);
printf("\t%d",c->num);
c=c->next;
}
}
```

e)

```
#include"ggc.h"
//test bench
void main()
{
int x,i;
stack_high_ptr=&x;
block rtemp;
void print_list(struct link *c);
struct link *head,*tempxyz,*a ,*b,*c;
size_t size_total=0;
a=(struct link *)GGC_malloc(sizeof(struct link));
a->num=0;
c=a;
for(i=1;i<10;i++)
{
c->next=(struct link *)GGC_malloc(sizeof(struct link));
c->next->num=10*i;
c=c->next;
}
c=NULL;
printf("\n&a=%0x",&a);
print_list(a);
c=a;
while(c!=NULL)
{
if(c->num==50)break;
c=c->next;
}
c->next=NULL;
GGC_garbage_collector();
printf("\nAfter collection\n");
print_list(a);
GGC_garbage_collector();
printf("\nAgain after collection\n");
print_list(a);
GGC_garbage_collector();
printf("\nAgain after collection\n");
print_list(a);
GGC_garbage_collector();
printf("\nAgain after collection\n");
print_list(a);
for(i=1;i<10;i++)
{
c=(struct link *)GGC_malloc(sizeof(struct link));
c->num=1000*i;
c->next=a;
a=c;
}
printf("\nafter adding some another node\n");
print_list(a);
}
void print_list(struct link *c)
{
while(c!=NULL)
{
printf("\t%0x",c);
printf("\t%d",c->num);
```

```
c=c->next;
}
}
```

B. *Metrics*

We have used following metrics for result analysis:

i. Allocation time (in 0-30ms)

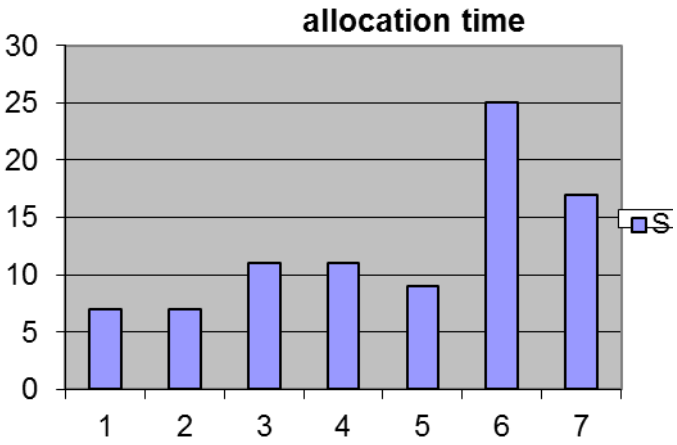

Figure 2. Allocation Time (in ms)

ii. Marking time

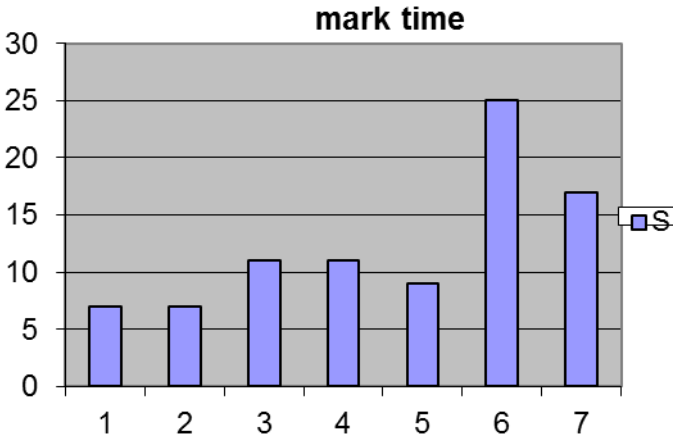

Figure 3. Mark Time (in ms)

iii. Total size of reachable objects

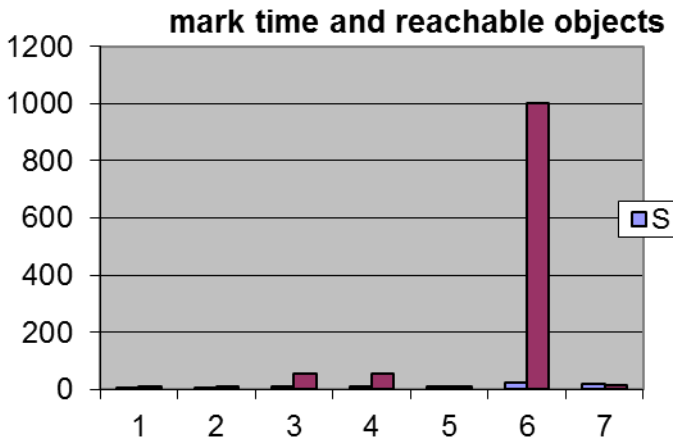

Figure 4. Mark Time (in ms) and reachable objects

iv. Allocation time for Boehm GC

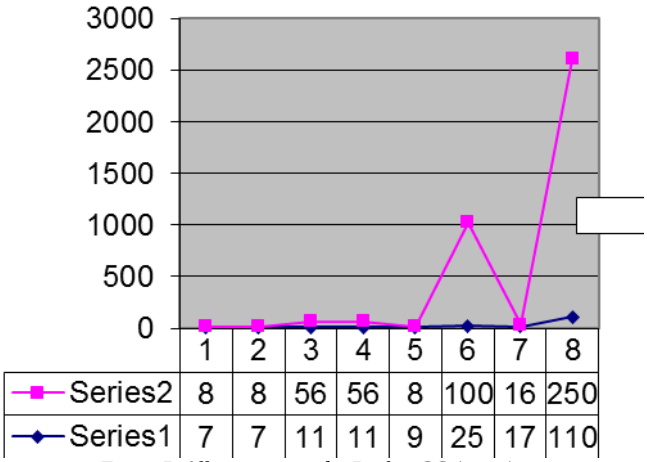

Figure 5. Allocation time for Boehm GC (in ms)

C. *Expected Outcomes*

i. Allocation time increases if size of block increases.
ii. Mark time depends on number of reachable blocks and size of each block.
iii. Copy time increases with size of block and complexity of Data structures.
iv. Compaction will be done in memory.

VI. Conclusion And Future Work

A new efficient algorithm is introduced for performing the garbage collection for dynamically allocated data objects and it is based on a conservative multi-generational age-based approach which is real time in nature. In this paper, we have given a small review of various garbage collection techniques and also presented a new garbage collection technique called conservative multi-generational age-based algorithm with fast allocation, suitable to implement for many object oriented languages. The proposed algorithm considers the third level multigenerational garbage collection of unreachable objects from live objects.

In future work, we can plan to improve the evaluation of this proposed conservative multi-generational age-based approach to compare with the other implementation of Garbage Collection techniques what are introduced earlier. The future work to reclaim the dynamically inserted data objects during the program execution can be extended if these data objects are found unreachable.

References

[1] Rifat Shahriyar, Stephen M. Blackburn and Kathryn S. McKinley, “Fast Conservative Garbage Collection”, OOPSLA ’14, October 20–24, 2014, Portland, OR, USA.

[2] A. Baldassin, E. Borin and G. Araujo, “Performance implications of dynamic memory allocators on transactional memory systems”, In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pages 87– 96, New York, NY, USA, 2015.

[3] S. M. Blackburn, P. Cheng and K. S. McKinley, “Oil and water? High performance garbage collection in Java with MMTk”, In 26th International Conference on Software Engineering, pages 137–146, Edinburgh, May 2004.

[4] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Sholten, and E. F. M. Steffens, "On-the-fly garbage collection: an exercise in cooperation", In Communications of the ACM, pages 966–975, 1978.

[5] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures. SIGARCH Comput. Archit.", News, 21(2):289–300, May 1993.

[6] R. L. Hudson and J. E. B. Moss, "Sapphire: Copying gc without stopping the world", In Proceedings of the 2001 Joint ACMISCOPE Conference on Java Grande, JGI '01, pages 48–57, New York, NY, USA, 2001.

[7] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer, "The collie: A waitfree compacting collector" In Proceedings of the 2012 International Symposium on Memory Management, ISMM '12, pages 85–96, New York, NY, USA, 2012.

[8] P. McGachey, A.-R. Adl-Tabatabai, R. L. Hudson, V. Menon, B. Saha, and T. Shpeisman, "Concurrent gc leveraging transactional memory", In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '08, pages 217–226, New York, NY, USA, 2008.

[9] F. Pizlo, E. Petrank, and B. Steensgaard, "A study of concurrent real-time garbage collectors", In Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, pages 33–44, New York, NY, USA, 2008.

[10] C. G. Ritson, T. Ugawa, and R. E. Jones, "Exploring garbage collection with haswell hardware transactional memory", In Proceedings of the 2014 International Symposium on Memory Management, ISMM '14, pages 105–115, New York, NY, USA, 2014.

[11] P. R. Wilson, "Uniprocessor garbage collection techniques", In Y. Bekkers and J. Cohen, editors, Proceedings of the International Workshop on Memory Management, volume 637 of Lecture Notes in Computer Science, pages 1–42, St Malo, France, 16–18 Sept. 1992.