I.J. Information Technology and Computer Science, **2016, 11, 8-15**
Published Online November 2016 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2016.11.02

Development of Real-Time Capability in Application Virtual Machine using Concurrent Automatic Memory Management Algorithm

Charan K V
Siddaganga Institute of technology, Visvesvaraya Technological University, Karnataka, India
E-mail: charanssit@gmail.com

A.S Manjunath
Siddaganga Institute of technology, Visvesvaraya Technological University, Karnataka, India
E-mail: asmanju@gmail.com

***Abstract*—**Superior garbage collection algorithms are needed for deterministic runtime system in complex embedded systems to explore the benefits of contemporary and conquered application programming language. Android embedded operating system is greatly used world wide as a mobile platform without denying this fact it also attracted researchers and engineers to integrate in other embedded real-time systems. It exploits Java language for embedded application development and it can also enhance a certain real time capability with the adoption of real-time support at Dalvik Virtual Machine (DVM). Need for Real-time garbage collection algorithms in embedded systems is identified by achieving new insights into the existing garbage collection algorithms through finding blemishes in it. The space based technique is used in proposed new Real-time GC algorithm for execution runtime system and Real time Garbage Collection (GC) schedulability issue is also addressed. The intuitive performance analysis result demonstrates reduction in the response time and also describes the determinism characteristic of the real time applications using proposed solution.

***Index Terms*—**Contemporary, virtual machine, real-time, complex embedded systems, schedulability.

I. Introduction

Dynamic memory management is a characteristic of runtime system in modern programming language implementation. It helps for high level of abstraction through automatic memory management to increase the software productivity and efficiency of software development. The automatic garbage collection algorithms for dynamic memory management during program execution largely influence on the predictive responsiveness of the system. Algorithms improvement is considered as part of improving the performance of Real-time systems since algorithm are also one of the discipline that affect the real time system engineering. The software is made by codifying the algorithms that intimately control the hardware. Research and development of algorithms is an art more than a science.

Computer system that perform a desired function or a group of functions by using a specific embedded software on a low hardware configuration with small memory footprint and low power consumption is called embedded operating system. Android, uClinux, windows CE, Symbian are few among the many embedded operating systems. The evolutionary improvement either in terms of features and supported hardware android is trying to surpass its origins and started migrating into new devices other than mobiles further creating rising interest in adopting android for embedded real-time environments few of the recent efforts already made [1,2] ensures the importance of it. The industry's movement from personal to embedded computing requires innovative mobile smart products combining many computing features to help for Bring your own device (BOD) and Internet of things (IOT) concepts. Noticing about utilizing android for embedded systems Karim Yaghmour [3] and gargenta has authored books for Android system integrators and programmers. Concurrency is becoming prevalent in more widely used Real-time embedded systems because of the multicores and larger heaps created by inexpensive RAM [15][17] . Real-time ability in concurrent garbage collection [18][19][20] is one of the enhancements for using advanced languages in many of the time critical systems.

The section II covers conceptual details of compile time, run-time environment in android DVM. It is one of the application virtual machine uses garbage collector in its execution engine. There are mainly two ways to integrate Real time support at VM level in android one is inclusion of another real time Java VM. There are many advantages from this approach and integration issues may also arise between the VM and kernel. Some efforts made [4] on using explicit memory management for the creation and freeing of objects to completely avoid GC pauses but the problem is developers can't free the objects that he doesn't created by himself (e.g. Android inner class methods for updating the user interface). Second is using Real time supporting garbage collection

algorithms at Runtime the proposed work tries to accomplish second method.

Section III introduces the garbage collection algorithms including its origin and classification with illuminating the drawbacks of different techniques. It also describes the need for Real-time Garbage collection techniques that will considerably increase determinism and reduces latency. The proposed system goal is to prevent the concurrent mode failure situations. Section IV is made up of proposed algorithm following result analysis and conclusion.

II. Dalvik Virtual Machine

In this section we are focusing on virtual machine running as a normal application inside an operating system for supporting a single process. Shi, Yunhe, et al [5] has mentioned about virtual registers. DVM uses registers based instruction set having virtual registers to store, manipulate operands and it also uses zygote process model similar to fork process of linux. The registers that dalvik bytecode refers to are not machine registers, but they are locations on the call stack. When method is called dalvik allocates enough memory on stack frame to hold all the registers that method needs. The VM load the values into a machine register in order to perform calculations, the results may be reserved in a registers to be used later without immediately writing it back to the corresponding stack location that requires push and pop operations to store intermediate values of the calculations. The values kept in the registers will be flushed back to the call stack only when it is needed. Hence virtual register machines have the potential to significantly reduce the number of instruction dispatches. Zeeshan I and Khan [6] reviewed on functionalities of android dvm. A dalvik is an extremely compact representation for an executable with devices having limited resources. DVM is cornerstone of the Android platform provides multithreading support and memory garbage collection on the platform. Wen Hu and Yanli Zhao [7] have made a research on process model of android DVM. Unlike conventional Java VM design, each instance of the DVM will not have entire copy of the core library class files and any associated heap objects.

Unlike standard Java Virtual machine (JVM) the jar files are not created instead of that DVM introduced another dex tool feature during compile time as shown in Fig 1. The dex tool is used to assemble two or more class files into one single dex file. The structure of dex file avoids the redundancy of the data and moderates the overall dex file size compared to larger jar files created by conservative JVM. Dex loader is used instead of class loader to load the created .dex file and prepare it for execution.

The dex files along with certain resources are packaged using android asset packaging tool (aapt) and apk builder is used for creating application package file (apk). The .apk file can be distributed easily and installed on any android supporting devices. Contrast to JVM, the DVM consists of single .dex file having shared method area for multiple java classes and methods in the classes. Every running application is assigned a separate heap space on the system physical memory however two threads of the same process can trample on each other's heap area. The application heap size is device dependent and depending on the device there is a hard heap size limit in android. Thread stack stores a thread's state in discrete frames called stack frames each frame encompasses of local variables area, operant stack, and Frame data having metadata of the respective stack frame. The Program Counter (PC) register is like a pointer to the current instruction. In the sequence of program instructions it keeps track of the instruction execution at any time this is same in DVM also. Inherent multithread support in java creates PC register for every new thread.

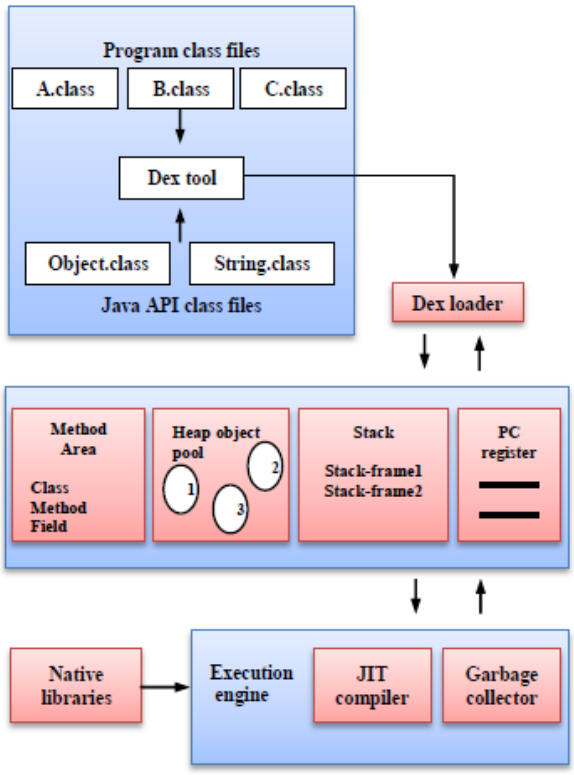

Fig.1. Conceptual structure of compile time and run-time environment in android DVM.

The Fig.2. is derived from the code analysis in Android Open Source Project (AOSP) it denotes the process of dynamic memory allocation to the application thread. The garbage collection is called with different reasons and varying kind of collections it perform. Concurrent garbage collection process is scheduled when your heap begins to fill up is called GC_CONCURRENT. GC_FOR_MALLOC is a simple mark sweep technique it is triggered when application attempted to allocate memory but already heap was full. The GC_HPROF_DUMP_HEAP occurs when you create a HPROF file to analyse the heap, it is only used for monitoring the heap usage. When you call System.gc() from an application the explicit garbage collection called GC_EXPLICIT follows. GC_EXTERNAL_ALLOC happens only on API level 10 and below for garbage

collection of externally allocated memory.

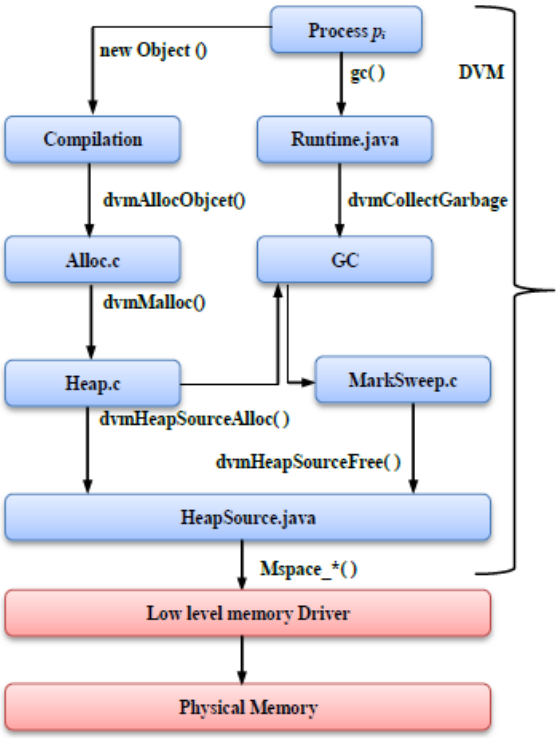

Fig.2. Memory Management in dalvik virtual machine.

III. Garbage Collection Algorithms

Since from the development of modern high level computer programming languages they are providing means to create objects. As long as references to the objects is existed by some series of pointer traversals starting from member of the root set these are called live objects. Most of the GC algorithms developed are based on the basic garbage collection techniques in [8] reported about this. If the program contains objects to which the references do not exist they are called Garbage objects that cannot be reached by traversing through any live objects. The process of finding reusable garbage objects and making it available for future allocation is called garbage collection. Early programming languages like pascal, C provides means to destroy an objects explicitly leads to two kinds of problems that will be found as hard to get right. One is dangling reference problem which arises due to early destruction of the objects they are still referenced by some other objects in a program. Second is memory leakage problem due to the too much delay in the destruction of garbage objects causes accumulation of garbage objects and memory to be exhausted easily. The explicit program memory management may provide programmer ability to explicitly control destruction of objects at any time. However it limits the software productivity because programmers find explicit memory management can be too costly for getting right amount of garbage collection and ensure objects destructed is neither too late nor too early.

Several approaches are exist for automatic garbage collection having different pause time, memory usage, implementation complexity, execution time, correctness,

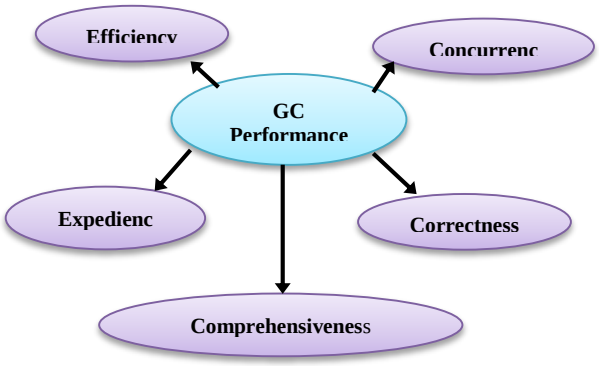

Fig.3. Garbage collector performance Metrics.

Comprehensiveness, robustness these design metrics will compete for one another [8] has identified the general goals in garbage collection schemes. The Fig 3 shows some of the factors influencing on performance of GC algorithm and varying values of the GC metrics leads us with a trade-off based on the suitability of the algorithm in particular platform. Some of the essential characteristics of ideal garbage collection algorithms is minimal overall execution time, optimal space usage, minimal pause time, improved locality for mutator unfortunately ideal scheme to fulfil all these goals is not possible in general. The fundamental Question of how to guarantee the worst-case pause time and how to maintain the regularity in pause time is addressed in the proposed algorithm by providing a novel method for garbage collection to ensure real-time support during application using automatic memory management.

A. Classification of Garbage collection algorithms

Garbage collection has been in use since its invention for the Lisp programming language that is reported in [9]. The smalltalk is the first language to use both object oriented programming and garbage collection. All GC algorithms are built upon three different key methodologies of garbage collection as shown in Fig 4. There are mainly two types of garbage collection techniques one is reference counting system in which each dynamically allocated object is associated with reference count representing the number of object references to the corresponding object. The second approach takes a global perspective on the aliveness property of the objects. It is brute force like approach of recognizing reusable objects, in this technique garbage collection problem is formulated as a graph problem and it is a more straight forward technique of finding aliveness of the objects using pointer traversals starting from some root nodes to all reachable nodes is called tracing method of garbage collection. The tracing method of garbage objects relinquishing is subdivided into copying and marking technique. Copying will eliminates the fragmentation problem by copying all live objects into to space and marking technique requires additional memory compaction step to alleviate the fragmentation

after marking the objects. The objects that are reachable are marked by either altering bits within the objects or recording them in a bitmap. Android uses bitmap marking to store mark bit's in a separate bitmap table, the size of the bitmap table is inversely proportional to the smallest object in the heap. Bitmap tables are expensive to access the mark bits but there are technically good reasons for using bitmaps in android one is mark bits held in RAM can be read or written without page faults. Second is no object or page is dirtied during marking because there is no swap disk to write back any pages. Third is a live object need not be touched during sweep. Fourth is objects live and die in clusters, so that 32 bits at a time. Finally less chance of program messing with bits also increases safety.

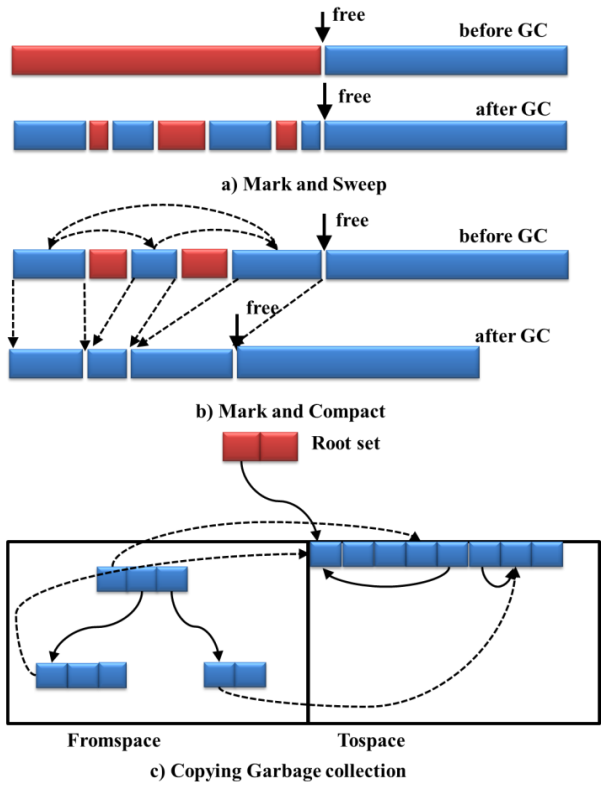

Fig.4. Key methods of GC.

Limitations of existing garbage collection methods for lack of support in real-time garbage collection.

- The Reference counting is not comprehensive because it can't reclaim cyclic data structure and also the support for any form of concurrency does not exist.
- The conservative mark-sweep is commonly called as *stop the world* garbage collection technique since it is a basic version of GC algorithm does not use concurrency leads to long zombie times for garbage collection.
- The CPU becomes idle more frequently in incremental garbage collection technique because for every allocation garbage collector will be called.
- Garbage collectors based on the object lifetime will creates memory regions for objects of different survival time as presented in [10] GC algorithm based on life time of an objects. This kind of dynamic memory management is known as Generational garbage collection it will not improve the expected pause time during worst case and also constrains the structure of the application for getting acceptable pause time.
- Copying garbage collection technique consumes twice as much of memory than the program actually requires and all mutator threads will be stopped during copying to avoid inconsistency.
- RTSJ (Real Time Specification for Java) is a variant of java designed for real time programming it uses new memory management schemes like immortal and scoped memory since Dalvik is not an RTSJ implemented virtual machine it is very difficult to use RTSJ for android java.

The Android Mark-Sweep algorithm presented here shows how simple Mark-Sweep is performed in Gingerbread and early versions of it. However entire garbage collection cycle will run at the cost of application pauses of around 500-1000 ms range. It is articulated in [11] that when heap size expands in android powered devices mark-sweep can be time consuming if it is applied to large heap area with lot of live data objects. Simple mark-sweep is stop-the-world kind of garbage collection technique this can blocks all mutator threads until the completion of GC. The panacea to this problem is found by developing many improvements for significantly reducing garbage collection pause time.

IV. Proposed Algorithm

Compared to existing garbage collection schemes [21][24][25] the proposed algorithm makes use of space based technique for scheduling the garbage collector. This technique will schedules and runs the collector according to the memory usage and availability of the memory in the heap as shown in Fig 5. Here there are two boundaries, one is depending on the initial available memory I and another one is depending on the previous available memory P_r since from the last GC cycle. The RTCMS is invoked when any of the two dynamically formulated boundaries will exceed. The circumscribe in scheduling the collector assures the enough heap memory before calling the collector. This is in contrast to scheduling GC_CONCURRENT process as shown in Fig 2 when memory is about to be exhausted. It is very much prone to the concurrent mode failure situations; the proposed schedulability condition will avoid this by ensuring more memory when GC is running. It enhances the real-time support in GC algorithm without constraints on the application virtual machine, programming language or need for any special hardware.

The proposed garbage collector design will gives generalised solution for avoiding the pause time length and reducing the inconsistency in GC pauses. It improves

deterministic program execution time in embedded systems where memory is considered as time. In the proposed approach the garbage collection schedulability is propositional to the availability of the space in heap and the amount of memory used since from the last GC cycle. The amount of work to be done with the creation of new objects by the mutator threads before calling the next GC cycle is constrained by the difference between previous availability and initial availability of the heap memory as well as currently available and previous available memory.

Step 1: Root scanning

Root scanning is one among the two major sources of blocking exist in any garbage collection algorithm it is a vital step and necessarily takes very minute amount of time as a part of every garbage collection. The pointers into the heap from stack will set up the root set and it must ensure consistent view of the root set to avoid incorrect reclamation of objects. Stack frames of respected mutator threads are scanned atomically by the garbage collection thread to assert the root nodes of the object graphs. This can be generalized by delegating work to the individual mutator threads itself. Research efforts are made in [12][13][14] on Real-time GC by proposing a solution to complicated issues in GC like root scanning and heap compaction and concurrent collection. The parallel scanning of the stack frames by the application threads and acknowledging to the GC thread void the atomic scanning however the pause time is not completely eliminated. The further investigations is needed on root scanning phase of CMS GC algorithm running in multiprocessor system to reduce time spent by the collector for scanning the root set.

Step 2: Concurrent marking

In this step the GC thread performs the tracing activity starting from the objects in the root set to find every reachable object allowing all the mutator threads continue execution concurrently with the GC thread. Update bit of the object is set when the references of the corresponding object are changed during this step. It is not appropriate to use another bitmap table for storing mark bits of updated objects we can store those objects in dynamically created update list.

Step 3: Final marking

All mutator threads are stopped for a while during remarking and objects stored in the updated object list are investigated to change the status of the mark bit.

Step 4: Concurrent sweeping

There are two main tasks of any garbage collection algorithm one is identification of garbage objects is also called as scavenging, the above three steps are used to perform scavenging with the help of object tracing method. Second is removal of garbage objects is called as evacuation that is to be executed in this step and compaction is a additional task to increase performance of garbage collection algorithm by mitigating as well as trying for eluding fragmentation. In order to satisfy predictability in response time of real-time systems using garbage collected languages the Real-time Concurrent Mark Sweep (RTCMS) is pursued in a CMS context but we have made special attention to how the collector will be called to carry out its work. The space based scheduling policy will execute the collector when heap space consumption reaches dynamically set threshold value.

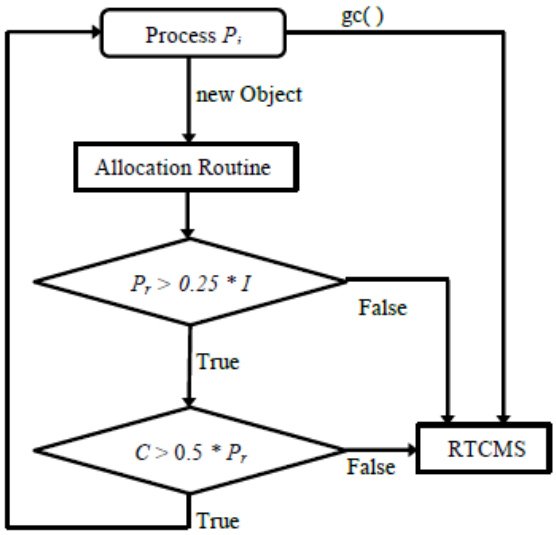

Fig.5. GC schedulability condition in RTCMS.

V. Results and Discussion

The implementation of various garbage collection algorithms is different from one another in terms of how they manage the heap. There are some simulation tools like a *gcSim* and *Glacier* [26] but they are not currently supported for concurrent garbage collection algorithms. We implemented the various garbage collection scheduling techniques using modern programming language. The random numbers are used to denote object allocations and object de allocations because these are random events. The reason we choose to do this is we could conduct studies on how variations of the number of object allocations and collections affect the performance of the given GC algorithms. The objects in heap are shown using matrix elements matrix M_1 is empty heap and matrix M_2 is full heap. Table.1. Results are obtained from our observation by executing different basic GC techniques. It shows decrease in the number of garbage collection scheduling pauses in proposed scheduling technique compared to other existing GC scheduling techniques. The results superimposed in Graph of the Fig 6 shows the comparison between time-based scheduling, slack-based scheduling and proposed space-based scheduling of the basic tracing collectors. This comparison also gives better understanding of how the different garbage collector scheduling policies will influence on decreasing the total GC pause time by guaranteeing sufficient mutator utilization. Minimum

mutator utilization (MMU) is the matric to measure mutator share of the processor in a given time interval.

$$MMU = \frac{T_c}{N_c}$$

$$M_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \cdots \end{bmatrix}$$

$$M_2 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & \dots \\ 1 & 1 & 1 & 1 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \cdots \end{bmatrix}$$

The above equation represents how MMU depends on pause time during program execution. N_c is the number of times collector is invoked. T_c is the total time collector is running concurrently in a given GC cycle. The value of T_c ranges from 1 to 10, the simple mark sweep algorithm will be assigned a minimum value and fully concurrent GC algorithm is assigned a maximum value.

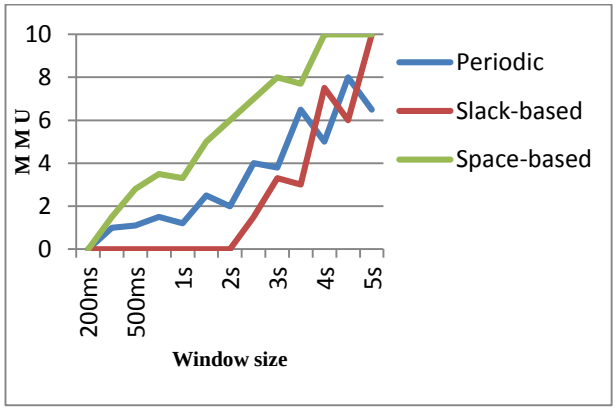

Fig.6. Minimum Mutator Utilization of different GC scheduling policies.

Table 1. Scheduling cost comparison between different GC Scheduling Techniques

Periodic		Slack-based		Work-based		Space-based	
Time	N_c	**Time**	N_c	**Time**	N_c	**Time**	N_c
20	5	20	2	20	45	20	1
50	11	50	4	50	98	50	4
100	20	100	6	100	126	100	7
200	31	200	F	200	182	200	11
300	40	300	F	300	235	300	14
400	51	400	F	400	290	400	18

$$N_c(\text{periodic scheduling}) = \frac{Window\ size}{GC\ period}$$

N_c for slack-based scheduling can be find out based on the number of times low priority GC thread is triggered when high priority thread will stop working. N_c of space based scheduling depends on number of times heap memory falls in the range $P_r < 0.25 * I$ *or* $C < 0.5 * P_r$. For a given window size of 5s and GC period of 1ms the calculated *MMU* for periodic scheduling policy is 6 due to the equally spaced GC pauses it is predictable with undesirable high cost of total pause time. The *MMU* for slack-based is 8.2 but it is unpredictable and space-based technique will bring *MMU* to 9.5 because of reduced N_c value. The T_c value of CMS is less than the RTCMS as a result of more possible concurrency failures in CMS. The RTCMS is diverging towards an ideal RTGC by avoiding likely occurrences of concurrency failure situations and mitigating N_c value. Consider the two same programs one running with CMS GC system and another one with proposed RTCMS GC system.

We also consider the case where concurrent system suffers from bursty allocation requests from many application threads it might cause failure situations. During program execution the runtime response from two systems certainly exhibit different execution time patterns with varying pause times having allocation rate of N objects per 10ms where the value of N varies depending on executing program. With all these assumptions the

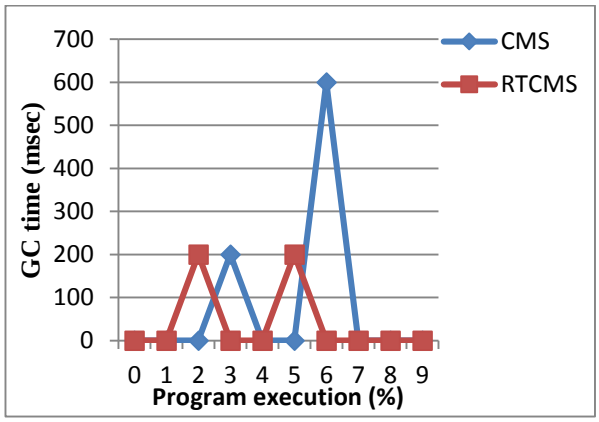

Fig.7. percentage of Program Execution vs GC time.

following graphs have been drawn intuitively with predictable outcomes for performing comparative performance analysis between CMS and RTCMS. Graph in Fig 7 shows the situation where RTCMS is superior to CMS. Mutator share of the processor is decreased as a result of concurrent mode failure at particular time slice in CMS compared to RTCMS. Timing diagram shown in Fig 8 provides good way to visualize timing properties of CMS, RTCMS and ideal Real-time garbage collector. The Regularity and consistency in pause time to obtain predictable timing is essential for qualifying to become ideal RTGC and pause time pattern of RTCMS in Fig 8 also shows consistency in pause time compared to CMS.

The proposed system gives more endurance against number of failure situations when burst allocations during garbage collection, this continues for a long time in worst case. It will defer the concurrent mode failure.

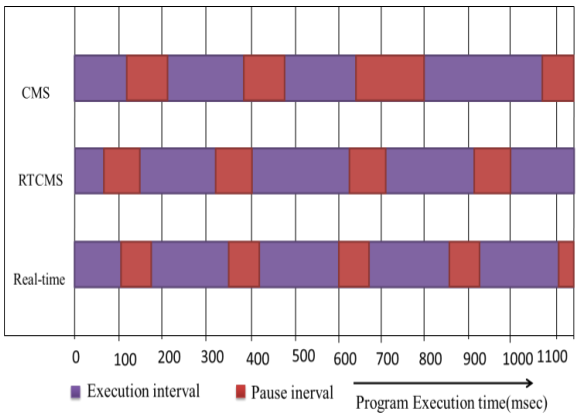

Fig.8. Timing pattern.

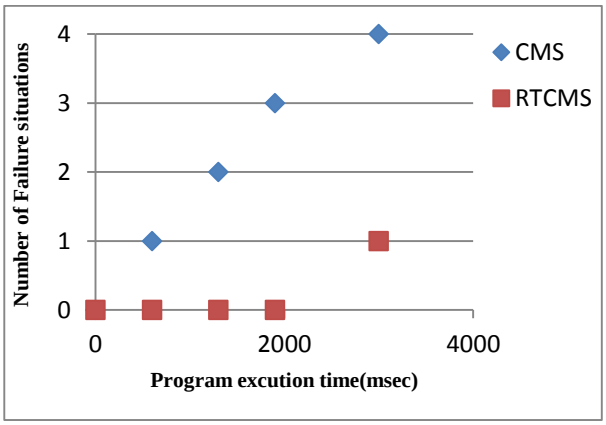

Fig.9. Program Execution time vs Number of failure situation.

VI. Conclusion and Future Work

Integration of RTGC algorithms in runtime system to obtain predictable program execution time is one of the important concerns in process virtual machine to make widely adopted modern programming language more suitable for real-time embedded systems. The birth and evolution of garbage collection algorithms is presented here and also takes a look at the runtime environment in android dalvik virtual machine. The runtime system research work carried out in this paper is for the development of RTGC algorithm by decreasing number of garbage collection pauses. The proposed algorithm mainly focused on garbage collection schedulability problem it is a challenging issue in dynamic memory management for hard real-time systems. Our future work involves analysing response time of multiprocessor system using space based RTCMS algorithm.

References

[1] Datta, Soumya Kanti, Sophia Antipolis, and France Soumya-kanti, Android stack integration in embedded systems*', International Conference on Emerging Trends in Computer & Information Technology, 2012*, Vol.7, No. 4, pp. 139-158.

[2] Yin Yan, Shaun Cosgrove, Varun Anand, Amit Kulkarni, Sree Harsha Konduri, Steven Y. Ko and Lukasz Ziarek. Real-Time Android with RTDroid, *MobiSys*'14 *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014, pp.273-286.

[3] Karim Yaghmour. Embedded Android *first edition O'Reilly publications*, 2013.

[4] Igor Kalkov, Dominik Franke et all. A Real-time Extension to the Android Platform', *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems, ACM*, 2012, pp. 105 - 114.

[5] Yunhe Shi, David Gregg, Andrew Beatty, M. Anton Ertl. Virtual Machine Showdown: Stack versus Registers' *ACM Transactions on Architecture and Code Optimization (TACO)*, 2008, Vol. 4, No. 4.

[6] Zeeshan I and Khan. A review on the functionality of dalvik virtual machine present in android operating system, *International Journal for Technological Research in Engineering*, 2014, pp. 627-638.

[7] Wen Hu and Yanli Zhao, Analysis on Process Code schedule of Android Dalvik Virtual Machine, *International Journal of Hybrid Information Technology*, 2014, Vol. 7, No. 3, pp. 401-412.

[8] Niels Christian Juul and Eric Jul. Comprehensive and Robust Garbage Collection in a Distributed System, *International Workshop on Memory Management*, 1992, pp.1–42.

[9] David F. Bacon Realtime Garbage Collection It's now possible to develop real-time systems using Java. *ACM new Applicative conference*, 2007, 3(24):305 - 307.

[10] Henry Lieberman and carl Hewitt. Real time Garbage Collector based on the Lifetime of Objects',*MIT Artificial Intelligence Laboratory*, 1981, Vol. 26, No. 6, pp. 419-429.

[11] Patrick Dubroy. 'Memory Management for Android Apps', *Google I/O, 2011*.

[12] Martin schoeberl and Wolfgang Puffitsch., Nonblocking Real-Time Garbage Collection, *ACM Transactions on Embedded Computing Systems*, 2010, Vol. 10, No. 1, Article 6, pp. 13-18.

[13] A. W. Appel, J. R. Ellis, K. Li. Real-Time Concurrent Collection on Stock Multiprocessors. *Proceedings of the ACM SIGPLAN'88*, Conference on Programming Language Design and Implementation, 1988.

[14] Bacon, David F., Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization, *Concurrency and Computation: Practice and Experience*, 2003 Vol. 23, No. 14.

[15] Dr Ali Ebrahim EI Desokey, Dr Amany Sarhan, Eng. Seham Moawad. Using Multiple Servers in Concurrent Garbage Collector. *Journal of Object Technology*, 2008 Vol.7, No. 4, pp. 139-158.

[16] Schoeberl, Martin. Real-Time Garbage Collection for Java, *Object and Component-Oriented Real-Time Distributed Computing, Ninth IEEE International Symposium*, 2006.

[17] Kalibera, Tomas, et al. A family of real-time Java benchmarks, *Concurrency and Computation: Practice and Experience*, 2011, Vol. 23, No. 14 pp. 1679-1700.

[18] Lorenz Huelsbergen, Phil Winterbottom. Very Concurrent Mark and Sweep Garbage Collection without Fine-Grain Synchronization', *International Symposium on Memory*

Management, 1998, Vol. 34. No. 3, pp. 166-175.
[19] Martin Kero, Johan Nordlander and Per Lindgren. A Correct and Useful Incremental Copying Garbage Collector, *Proceedings of the 6th international symposium on Memory management. ACM*, 20017, pp 129-140.
[20] Martin Kero. Garbage Collecting Reactive Real-Time Systems, *Lund University of Technology, Department of Computer science and Electrical engineering, EISLAB*, 2007, ISSN:1402-1757.
[21] Roger Henriksson (1998) Scheduling Garbage Collection in Embedded Systems, *Ph.D. dissertation, Lund University.*
[22] Robin Milner. Communicating and mobile systems: the Π calculus, *Cambridge University Press, 1999.*
[23] E. W. Dijsktra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens, On-the-Fly Garbage Collection: An Exercise in Cooperation', *Communications of the ACM* , 1978, Vol. 21, No. 11, pp. 966-975.
[24] Tomas Kalibre, Filip Pizlo, Antony L. Hosking, Jan Vitek. Scheduling Real-time Garbage Collection on Uni-Processors, *30th IEEE Real-Time Systems Symposium*, 2011, pp. 81-92.
[25] Wilson, Paul R. Uniprocessor garbage collection techniques, *ACM Computing survey in Memory Management*, 1997, pp. 527-540.
[26] Bruno Dufour, Glacier: A Garbage Collection Simulation System, Sable Research Group McGill University.

Authors' Profiles

Dr A.S Manjunath received his Ph.D degree from Bangalore University, India in 2001 This author is Rashtriya Rattan Awardee in 2004 for outstanding individual achievements and distinguished services to the Nation. He is working in the potential as a professor in C.S.E dept, S.I.T, Tumkur and also C.E.O of the ManVish eTech Pvt Ltd company. He is having 30 years of experience in industry and academics the areas of expertise include embedded systems, RTOS, Networking, Embedded Linux etc.

Charan K.V received Bachelor of Engineering degree in 2010 from V.T.U, Balguam, Karnataka, India and currently pursuing integrated (M.Tech.+PhD) dual degree at siddaganga institute of technology Research Center, Tumkur under Visvesvaraya Technological University, Belguam, Karnataka. His interested areas in research is operating systems, embedded systems, real-time computing.

How to cite this paper: Charan K V, A.S Manjunath, "Development of Real-Time Capability in Application Virtual Machine using Concurrent Automatic Memory Management Algorithm", International Journal of Information Technology and Computer Science(IJITCS), Vol.8, No.11, pp.8-15, 2016. DOI: 10.5815/ijitcs.2016.11.02