THESIS REPORT

On

Garbage collection in real time systems

For partial fulfilment of

MASTER OF TECHNOLOGY

IN

VLSI DESIGN & CAD

By

Amandeep kaur

(600861001)

Under the supervision of

Dr. Sanjay Sharma

Associate Professor

Department of Electronics & Communication Engineering

THAPAR UNIVERSITY

Patiala-147001, India

DEC 2009

DECLARATION

I hereby certify that the work which is being presented in the thesis entitled, ***"GARBAGE COLLECTION IN REAL TIME SYSTEMS"*** *in partial fulfilment of the requirement for the award of degree of M.Tech (VLSI Design & CAD) at Electronics and Communication Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Dr. Sanjay Sharma, Associate Professor, ECED.*

The matter embodied in this thesis has not been submitted in any other University/Institute for the award of any degree.

Amandeep Kaur.

Date: 8th July 2010

Amandeep Kaur

Roll No. 600861001

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

Dr. Sanjay Sharma
Associate Professor
ECED, Thapar University
Patiala-147004

Counter signed by:

Dr. A. K. Chatterjee
Professor and Head
ECED, Thapar University
Patiala-147004

15.7.10

Dr. R. K. Sharma
Dean (Academic Affairs)
Thapar University,
Patiala-147004

16.7.10

ACKNOWLEDGEMENT

Words are often too less to reveals one's deep regards. An understanding of the work like this is never the outcome of the efforts of a single person. I take this opportunity to express my profound sense of gratitude and respect to all those who helped me through the duration of this thesis.

First of all I would like to thank the Supreme Power, one who has always guided me to work on the right path of the life. Without His grace this would never come to be today's reality. This work would not have been possible without the encouragement and able guidance of my supervisor, Dr. Sanjay Sharma, their enthusiasm and optimism made this experience both rewarding and enjoyable. Most of the novel ideas and solutions found in this thesis are the result of our numerous stimulating discussions. Their feedback and editorial comments were also invaluable for the writing of this thesis.

No words of thanks are enough for my dear parents whose support and care makes me stay on earth. Thanks to be with me.

At the end, I would like to thank all the faculty members of the department and my friends who directly or indirectly helped me in completion of my thesis.

ABSTRACT

GARBAGE COLLECTION IN REAL TIME SYSTEMS

By Amandeep kaur

Automated memory management techniques on garbage collection, reduces the complexity and problems of manual memory management. When used properly, it will lower development costs and save time by eliminating the need to test the product for faulty memory managed code. With the size and complexity in today's systems, this is becoming increasingly crucial.

Even though this thesis addresses the garbage collection in general, it is specially biased towards the real time garbage collection and an implementation.

In this thesis the fundamentals of the three garbage collection techniques are surveyed. The fact that most, if not all, of the specific implementations that have been made are only suitable for interactive and soft real time systems makes this tasks server. Although, when these basic techniques are made incremental they become eligible for hard real time systems in the theory.

An implementation in java has been made that at least the hard real time requirements of predictability and schedulability. When the suggested changes and the modifications to the compiler is implemented, the implementation should be quiet efficient.

INDEX

LIST OF FIGURES

CHAPTER 1

Introduction

The comprehensive goal of this report is to analyze the opportunities of existing garbage collection techniques towards an embedded collector with hard real time requirements. Firstly a survey of the existing techniques was done. In its wake a survey of the fragmentation problem was done. Secondly, a software model of the garbage collector for simulation and testing was made. Since this report is oriented towards the examination of different techniques the software model is biased towards validating that it works rather than towards optimal performance.

It is beyond the scope of the report to perform static analysis. Such as a complete system analysis of robustness, efficiency, predictability and schedulability.

1.1 Automated Garbage Collection

The need of automated garbage collection, or automated memory management, in terms of time and memory is oblivious. If used properly it will cut development time in projects, the bigger and more complex project, the more time and time is money.

The ultimate garbage collector, or automated memory management scheme, should allocate the exact amount of memory needed when it is needed. It should also reclaim memory the moment it becomes useless to the running program. Furthermore, it should not interfere with the execution of the running program nor steal so much cpu power that it becomes the heaviest task performed by the cpu. All these demands together are of course a utopia.

Even though it is a utopia it is still what it shall accomplish. What needs to be done is to evaluate the different techniques that exist and perhaps, if needed, look beyond them to find a compromise that is sufficient for the environment in the system in question. What the environment might be differs a bit but to embedded systems it is usually real-time characteristics that are the key issue. More on this in the next section.

The basic idea in this report will be that the idle time of the cpu should be spent on relieving the different processes of garbage work when possible. By doing so the future analysis like static, memory and timing will become much more easier than if it were done together.

1.2 Real –Time Systems

Systems can be divided in to two categories. Those who only have to produce a valid result and those that has to do it in a specified time, a deadline. The bulk of the first kind of systems is called interactive systems. These are often user dependent and await input from the user. In this case, the response time is not crucial as long as the average response time is similar from time to time. The user will then learn and come to terms with the fact that certain operations take a certain amount of time. If it is not, the user may try repeatedly to get a response. This might leads to problems or at least irritation. The remaining systems of the first kind are systems like compilers.

The other type of systems is real time systems. All real-time systems do not share the same requirements. One needs to realize the difference between soft and hard real-time systems. The difference is that soft real-time systems can miss occasional deadlines but were as a herd real time systems can not because they may crash and loss of lives and/or money. An audio streamed is an example of a soft real-time system. If a deadline is missed in the stream the output will be affected but the stream will continue. Although with some what lower quality.

Hard real time systems can be found in medical equipment and other time important products such as guidance systems. These are systems which combine soft and hard real-time requirements, i.e. one thread have soft requirements and one have hard requirements, as an example see Henriksson {8}.

1.2.1 Real-Time Requirements

There are four attributes it is important when talking about real-time requirements. These are predictability, schedulability, efficiency and robustness, Wilson[?,Unipro]

Two of them, predictability and schedulability, are more important than the others. But the other two, efficiency and robustness, are very nice to have if not necessary, The first two goes hand in hand with each other, if the system is predictable it is easy to schedule, provided that it can be scheduled, It is more important to have a predictable system that having the fastest average system. Because the worst case execution time (WCET) on the fastest average system could fail to meet its deadline, thus failing its purpose and

therefore corrupting the system. It is because of this the efficiency criteria are of minor importance.

1.3 Structure of this Report

Firstly the report starts of with the fundamentals of garbage collection. After that comes a chapter addressing the fragmentation /allocation problem. This chapter ends with the choice of technique for the implementation and the motivation thereof. Following that us the chapter about the implementation. The theoretical aspects of the implementation and the differences to the actual today software implementation. Finally comes the summary. The chapter and the thesis is summed up with the conclusions.

CHAPTER 2

The Fundamentals of Garbage Collection

In this chapter the general idle of the three basic techniques will be presented. For further and a more exhaustive presentation see Wilson[24]. At the end of its chapter there is a section about more advanced solutions using these techniques. Specific implementations are in most cases left out since the intention of this chapter is to give the reader insight in this topic. All automated garbage collection has its origin in one of these and they can be combined to cover up each other cons.

The basic Garbage collection can be divided in to a two-part operation. The first one determines live objects from garbage and the latter one does the reclamation and reuse if the implementation supports reuse.

A live object is the object o the heap that is going to be used by the run-time program at least one more time, i.e. read this value.

To determine which object that live there are two methods. either by reference counting or by tracing. Included in the term tracing are both marking and copying techniques. As the name suggests reference counting keeps track of how many references thee are that currently pointing it each specific object. This information is used either exact or as a local approximation to determine true liveliness. Tracing collectors used root scans to traverse down into the heap to reach all live objects. Thus determines the liveliness more directly.

The reclamation step is further explained in each section since it is more technique specific.

2.1 Conservatism

In a perfect world the garbage collector would reclaim every object the moment the object becomes garbage. Since this is not plausible in general, the exact moment the object becomes garbage can not be determined, a garbage collector uses a reasonable approximation to when this moment occurs. This behavior introduces conservatism in the

garbage collector. The garbage collector can not gamble and reclaim objects at will, it has to be conservative and know that this object is not going to be need again.

All garbage collectors have a bit of conservatism in them. Reference counting garbage collectors are conservative to logically and tracing garbage collectors introduce a major temporal form of conservatism. This due to the fact that they allow garbage to float between garbage cycles. The degree of conservatism differs from implementation to implementation and gives different performance tradeoffs:

2.2 Reference Counting

The straight forward approach of reference counting is quite simple. Each object is equipped with a counter that keeps count of the references (pointers) to it. When the object is created the counter is incremented. It is also incremented each time a new reference points to the object. Each time a existing reference is eliminated the counter is decremented. When the counter reaches zero the memory occupied by the object is reclaimed. This since the object have no more references towards it and therefore e running program can not access it. The counter and some other information, typically a description of the object, is fitted in a header of the object. This header is not visible at language level and not in any of the illustrations in t is thesis, unless if there is a certain point to be made, see figure 2.1.

If a reclaimed object is the top object of large structure a problem emerges. This since all pointers, in the object, to other objects are examined and the new objects counter is decremented. That could lead to a transitive decrementing of reference counts and reclaiming many other objects. Which means that all of a sudden the garbage collector has to do lot of work. This will compete for cpu time with the running program. An easy solution to this is to add objects whose counter have reached zero to a list and process that list when there is time. This list will contain free but unprocessed objects.

In figure 2.1 there are the normal list f free blocks and a temporary list containing these semi-free blocks. All references to object A has just been eliminated, making As reference count zero. Thus, A is moved from the heap to the temporary list. A still holds references objects on the heap, this is why A is only semi-free. Even if there were no references from A to the heap A would have been moved to the temporary list. T s since it

is costly in time to examine the content of A. This cost is moved forward in time, to be taken during an period were system is idle for example.

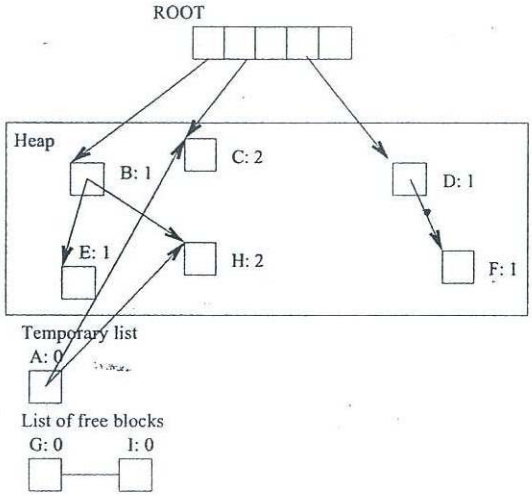


Figure 2.1: Small example with temporary to be free list 1.

As time goes on the references to object B has been eliminated, see figure 2.2 b is linked to A, waiting for the real deallocation. Since there still are objects in the list of free objects, G and I, there are no need to tamper with the temporary list during critical runtime. If the list of free objects should be empty when an allocation occurs, objects from the temporary list will be used. Although not before they are deallocated in an orderly fashion.

When the moment of deallocation arrives. The first object in the temporary list, A, is examined for references to other object on the heap. Since A had two references on the heap, objects C and H, their reference count will be decreased. When that is done A is added to the list of free objects. In figure 2.3 object B is undergoing the examination for references to other objects. Since object E is found and only had a reference count of one, E is moved to the temporary list.

When the problem of rampage freeing is solved with the temporary list reference counting is inherent incremental. Each normal reference count adjustment contains only a few operations and every operation is closely linked with the process and object that created them. is locality distributes the cost of garbage collection to the object or process that created the garbage. The cost is proportional to the amount of pointers in he object or

process. Another thing is that the idle process or a low priority process have to take care of the processing of objects in the temporary list.

With this setup it is easy to satisfy real- time requirements, guaranteeing that the memory management does not interfere with time critical tasks or processes.

Another positive thing is that there is not any penalty if the amount of live data is close to the amount of total memory. The performance is essentially the same which is important in embedded systems with very small amounts of memory.

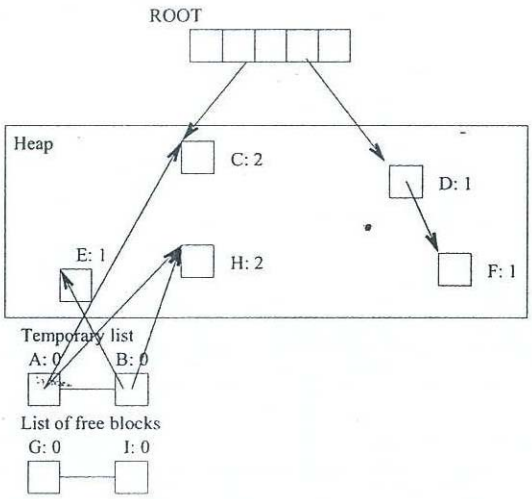


Figure 2.2: Small example with temporary to be free list II.

2.2.1 Problems

All is not well with reference counting, there are three major problems, cyclic garbage, fragmentation and efficiency. The first one is a big problem if cycles occur in the memory. A cycle is a directed graph, see the right part of figure 2.4. When the reference from the root to object A is eliminated at time X + Y the cycle between object A and object C will t be reclaimable and the memory segment that object A and object C occupies will be lost. There are some manual and automated techniques to solve the e problems, see Ritzau[18].

External fragmentation can be avoided at the cost of internal fragmentation. This can be done if a fixed block size s set. Internal fragmentation is not fun but external is worse, internal fragmentation is predictable and therefore one can live with it. It is a waste of memo but the other garbage techniques also waste memory, depending on

implementation, much more memory. More on how to set the block size and different techniques in the next chapter, fragmentation/ allocation.

The last problem, efficiency, is that the running program has to increment and decrement all references. If a pointer changes object, it points to another object, the cost is twofold. This since the first object has to be decremented, checked if its counter reaches zero and the there object has to be incremel1ted.

Another overhead is short lived stack based variables that occur quite often. Since the variable is so short lived the value of the reference counter will fluctuate a lot. This overhead can be reduce with static garbage collection in the form off among many analysis strategies pointer and escape analysis. Which only increments a counter if the point escapes its function. A deeper presentation about static analysis, see Park and Goldberg[15][14], Salcianu and Rinard[19] and Detlefs[6], is left out because a the boundaries of this thesis.

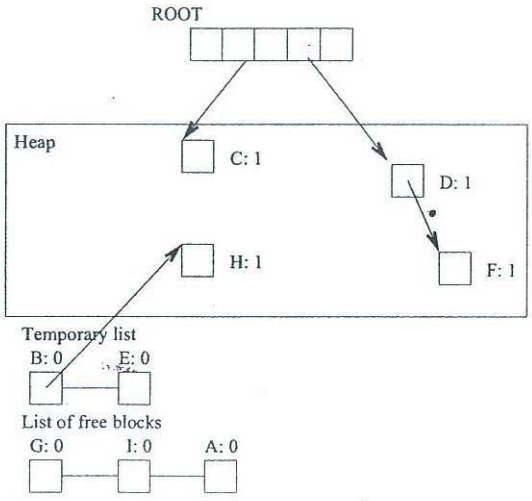


Figure 2.3: Small example with temporary to be free list III.

2.3 Mark and Sweep

The first step of a garbage cycle when using to distinguish the live objects form the garbage breath-first or depth-first search algorithms f has been determined to be live the object is m setting some bit or updating some table.

After that the heap is swept to eliminate any garbage. The garbage that is found is usually fitted into a doubly linked free list, just as in reference counting. When this is done the garbage collector is ready to start all over again in a new garbage cycle.

2.3.1 Problems

There are two problems with traditional non- incremental mark and sweep algorithms. These are external fragmentation and the cost of the sweep. The first problem is due to the fact that when a object has been allocated it uses that space in memory until it becomes garbage. That means that if objects

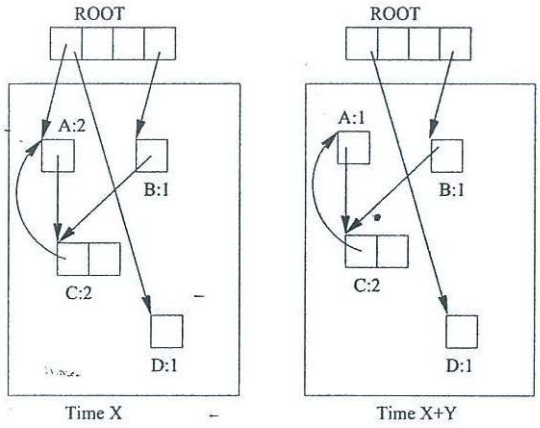


Figure 2.4: Reference counting and the problem with cyclic garbage.

have different sizes a problem with external fragmentation will arise. But as in the case of reference counting it should be possible to solve if there were a -fixed block size, more on that in the next chapter, fragmentation/allocation. Another solution is to complicate the garbage collector by making it generational, so that old objects are moved or given a specific part of the memory to reside in.

The cost of the sweep is proportional to the size of the heap. Due to the fact that all parts of the heap has to be searched for marked live objects. This cost can be reduced for the cost of comlexity and memory. A generational implementation could for example search only the part of the memory were the youngest objects reside every garbage cycle. The memory cost comes from the fact that the memory has to be divide into different areas for the different generations. Which reduces the utilization of the memory.

These problems can be dealt with by using different amount of complexity when implementing the algorithm. Making the garbage collector generational is only one of

many methods, see the sec on Generational Techniques later in this chapter and Ritzau [18] for a couple ore.

2.3.2 Mark and Compact

This is a variation of the mark and sweep garbage collection that solves the fragmentation problem. The difference lies in the second phase when all live objects are found. Then the live objects are compacted eliminating external fragmentation. This is done by sliding or copying objects to form a continuous chunk in the heap. In figure 2.5 the thought is to illustrate a copying compaction using breath-first. Firstly object A is moved to a corner of the heap, after that object D has been moved. The ext object scheduled for movementis B. The ordering of objects in the chunk is implementation dependent, meaning that this example should not be seen as the only right way to do it.

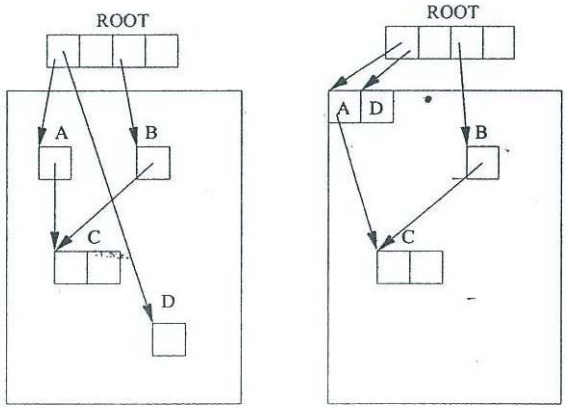


Figure 2.5: Example of mark and compact.

Unfortunately, the mark and compact algorithm requires more passes over the memory than mark and sweep which makes it significantly slower. There are several passes after the initial marking phase to among other things compute the new locations for the objects and moving them. For a more exhaustive presentation see Cohen and Nicolau[4].

As with mark and sweep there are a lot of variations to this technique. One could use the two-pointer algorithm, create by Daniel Edwards, where one pointer starts at the top of the heap looking for live objects. The other one starts from the bottom looking for holes. In addition to this one there are some where the memory is divided to fit special sizes of objects.

2.4 Copying Garbage Collection

As stated earlier copying garbage collectors belongs to the family of tracing garbage collectors. Just like mark and compact garbage collectors copying garbage collectors move the live objects to avoid fragmentation. But unlike mark and compact garbage collectors copying garbage collectors do not need several passes over the heap, thus saving quite a lot in speed. This is accomplished with integration between the marking phase and the copying process. When a live object is found it is copied to an area of the heap that is known to be free and all the pointers to and from the object is updated.

The usual approach is to use semi-space to be able to know that there is an area that is available. This means that e memory is divided into two different segments. The segments is called fromspace and tospace. During

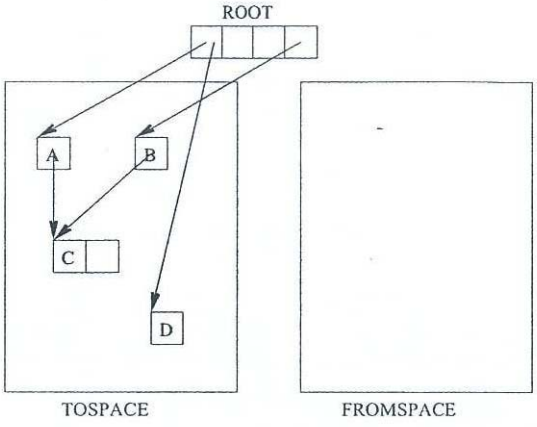


Figure 2.6: Copying garbage collector before garbage cycle.

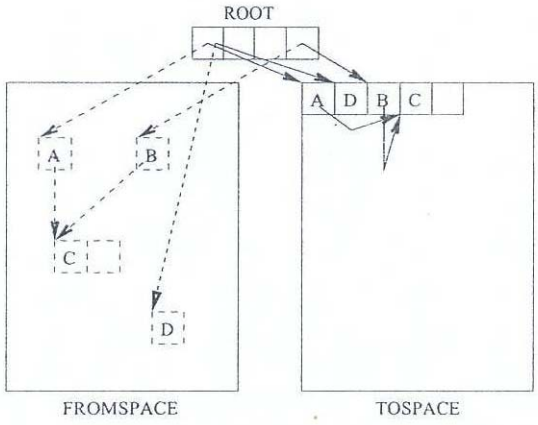


Figure 2.7: Copying garbage collector after garbage cycle.

normal program execution only one of the is in use, see figure 2.6. The first thing that happens in a garbage cycle is that switch takes place. The current tospace becomes

fromspace and the current romspace becomes tospace. After that, the traversal from the root set begin When a live object is found the object is copied from fromspace to the beginning of tospace. Where the object ends up in tospace is implementation dependent but the beginning or end of tospace are good placements. When al live objects have been found and copied they form a continuous chunk in to ace, see figure 2.7.

This is the technique that the Scottish company Linn[7] used in their processor rekursiv. The processor was originally made in the 1980s and is mostly hardware implemented.

2.4.1 Problems

The biggest issue is the fact that only half t e available memory is active at any time during normal runtime. This is no a problem if there is an unlimited amount of memory, then the collector could not be necessary, but there is not and if the system is embedded the amount is usually very small and expensive to extend. On the other hand theJ1 is a natural inheritance that no fragmentation will occur, neither internal no external.

Another thing is the cost of the copying. If there are large objects then it is going to take a long time to copy them. Doing this time interrupts are not welcome, i.e. it is impossible to interrupt the copying process.

Suppose that the amount of live data is almost the same as half the total amount of memory (one half fromspace and one half tospace). Then the garbage collector has to run quite often and each time it has to copy all live objects to tospace. This can be solved by add g additional memory.

2.5 Making the Garbage Collector Incremental

In order to enjoy and make use of real-time characteristics the garbage collector has to be incremental. This since it is not acceptable that the execution should be halted due to the fact that the garbage collector has to free dead objects resident in memory. Therefore the cost of garb e collection must be spread out in small interruptible units so that there always will exist free memory when it is needed. This assumes that the amount 0 live memory do not surpass the total amount of memory.

Reference counting is, because of its inherent locality, quite incremental from the beginning and it is easy to make t fully incremental. -Due to the fact that reference

counting have some drawbacks, efficiency and cycles, most previous work have therefore gone into making tracing garbage collectors incremental.

When it comes to tracing garbage collect s the task of making them incremental is a bit more complex. In an incremental tracing algorithm there is an issue with coherence, multiple readers and e or multiple writers. More conservatism has to be added so that the garbage collector does not reclaim live objects. It does not matter if the reachable graph that the garbage collector sees is the correct one as long as all live objects survive the garbage cycle. To be able to interrupt a tracing garbage collector when t its tracing out the graph a read and/ or write-barrier has to be implemented That protects the garbage collector while in a garbage cycle and lets the running program mutate the graph according to predetermined conservative rules. For this reason, discussions of incremental tracing garbage collectors typically refers to the running program as the mutator. In other words the mutator ay alter the graph but only if it does it so that the garbage collector does not miss or forget any objects.

Since the reachability graph that the garbage collector sees has to be a conservative safe approximation of the real graph all garbage will not be collected. It will float until the next garbage cycle. This is an unfortunate drawback not to be able to reclaim memory right away b much memory. It is a small price to pay to avoid very expensive synchronized coordination between the garbage collector and mutator.

2.5.1 Tricoloring for Tracing Garbage Collectors

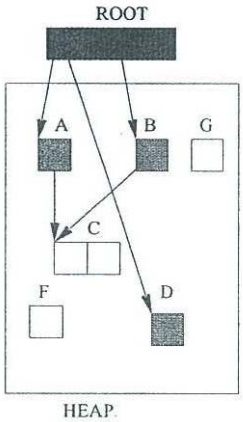


Figure 2.8: Tricoloring with a mark and sweep algorithm.

The tricoloring approach is actually m re of an abstraction that makes it easy to understand tracing incremental garbage collection. The process can be described as traversing- a graph and co ring the nodes as one pass them. There are usually three colors, black, grey and white. The black represents visited and examined nodes, the grey represent found but not examined nodes and the rest is white.

Figure 2.8 shows a mark and sweep garbage collector that has begun its traversal of the graph, marking objects A, and B grey since they have contact with the root set. The thought here is that n matter what the mutator does the garbage collector should when it is running just proceed with its traversal in an wavefront manner. Continuing until all objects in figure 2.8 except object F and object G, since they are garbage that c n not be reached, is colored black. While mark and sweep garbage collector sets a mark bit when it color an object a copying garbage collector colors its objects black when they move from fromspace to tospace. It should also be said that the gray objects in a mark and sweep garbage collector often are represented by the stack or queue of objectsused to control the marking traversal. In a copying garbage collector the grey objects represent objects in tospace that has not been scanned yet.

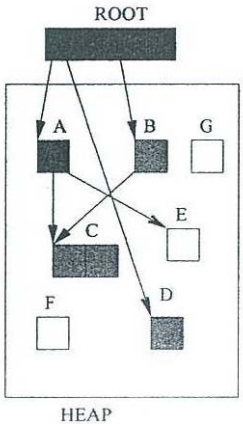


Figure 2.9: Later in the tricoloring(violated) phase.

These set of rules makes it a violation if black node have direct contact with a white node, see figure 2.9. The pointer from object A to object E will result in a violation since the garbage cycle is finished object E will be considered as garbage and therefore removed. To cope with this situation a read- and/ or write barrier must be implemented.

2.5.2 The Variations

There are three basic variations how to not violate the tricoloring unity. These are write barrier, read barrier or both. Though usually it is sufficient to have only one of the barriers. The barriers work as a coordinator between the garbage collector and the mutator. The area barrier detects when a mutator is trying to access a pointer to a white object immediately coloring that object grey. This since the mutater cannot read pointers to white objects.

In the case of the write barrier there are of different categories, depending on how the problem is viewed. In order to c n the garbage collector the mutator has to write a pointer to a white object i to a black object and destroy the original pointer before the collector sees it. f either of these conditions does not hold then there are no problems. The object in question will be reached later in the garbage cycle or treated like garbage since it is garbage.

The first approach is snapshot-at-beginning algorithms, they make sure that the second condition does not hold. These algorithms take a snapshot at the beginning of every garbage cycle thus saving the structure of pointers so that the collector can find all objects.

Incremental update algorithms are a bit more subtle and focusing more on the pointers that cause the problem. If a mutator moves or constructs a pointer from a black object to a white object, the mutator have to make the black object grey. Or at least parts of it grey, thus undoing some of the garbage collectors work. This ensures that the white object will be found later on in the garbage cycle. .

2.6 Further Development 0f the Basic Techniques

In this section there will be some example of how the basic techniques can be evolved to make up for shortcomings and or to improve performance.

2.6.1 Deriving Residual Reference Count

This technique makes the reference counting garbage collector more efficient. It is done by partly evaluating the program in 6rder to reduce the inherent run-time overhead of reference counting. Furthermore, it generates run-time detected selective updates on

recursive d a structures wherever possible. For further reading and the proof see Schulte [20].

It still does not solve the problems with cyclic garbage and fragmentation that comes with reference counting garbage collectors. However it if not solves, takes some burden of the efficiency problem, but at the cost of complexity.

This is partly compile-time optimizations, which means that the developer has to have access and right to alter the compiler. Furthermore, they are rather expensive in terms of compile-time. Since he order of complexity of analyzing a single function is at least exponential in he arguments of the function being analyzed, see Bloss [l].

2.6.2 Generational Techniques

Generational techniques applies to tracing garbage collectors. The idea is based on that since most objects die young and hose who survive tend to do so for quite some time. There should be a difference between early mortality and more long lived objects. Thus making it less cumbersome during heap compaction.

The memory is divided into areas, the number of areas is optional see figure 2.10, which are garbage collected individually. These areas are sometimes called generations and the process of ma age these areas is sometimes called tenuring. The age or rank is determined y the number of garbage collection cycles the object has survived. Depending on where the number of cycles is drawn, when a object passes the preset number the object is bumped one age or promoted one step.

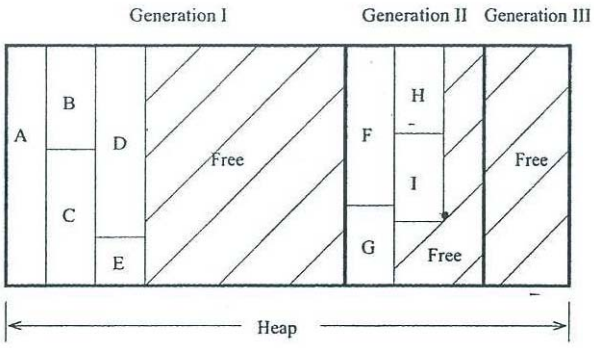


Figure 2.10: Mark and sweep with simplified generation.

In this means that the area with the young objects, generation I, has garbage collection done quite often. The other areas a garbage collected more seldom the older its inhabitant are. Thus making the ark and sweep garbage collector cheaper since it sweeps less memory over 11.

This makes the bulk of the moving operations in compacting and copying disappear since objects that live long is not affected by the garbage cycle in the young generation. Hence the average-case is decreased.

When using small amounts of memory even more memory can not be used all the time. Unless the amount of memory' the different areas is dynamic, which adds a lot of complexity if even possible in all systems.

Another thing is that the garbage collector has to bee quite general. A lot of care has to be devoted to ensure that the parameters of the garbage collector works out and maximizes the performance. Among the parameters there are four that stands out besides the others, these are:

- Advancement policy. When should an object be eligible for promotion to the next generation?
- Heap organization. How many generation should there by? And among these generations, how should the memory space be divided?
- Collection scheduling. For non increment garbage collectors, how might disruptive pauses be avoided or migrated? How often should garbage collection cycles run in the different generations? If the Different garbage collection cycles run in an optimal ratio garbage be reduced in incremental garbage be reduced in incremental garbage collectors?
- Intergenerational references. This technique builds on the possibility that garbage collection can be done on younger generations while the old lies dormant. That implies that live pointers from older generations into the younger one that is being collected must be found or provided somehow.

Returning to figure 2.10, if object A and B survives their garbage cycle and are eligible for promotion there will not be enough room for them in generation II. If it is a stop the world system then the garbage collector for generation II starts and the problem is more

or less solved. But this scenario should of course never happen in a real time system but if it do there has to be some way to recovery from it, i.e. to promote ob object in generation II or not promote object A and/ or B.

So for real time systems it has to be general or there has to be quite severe control of the programs which it runs with. Making sure that the amount of survivors each garbage cycle is not to high

2.6.3 Replication Garbage Collector

A variation of the copying garbage collector is the replication garbage collector. A replication garbage collector uses fromspace invariant which require that the mutator use only the original fro space objects. Thus the flip is made in the end of the garbage cycle instead of he beginning and the mutator continues to see the objects in fromspace until the flip. This eliminates the need for a read-barrier, although the need of a rite-barrier is introduced.

Nettles and OToole [12] has made an r I-time implementation of a replication garbage collector.

The downside in this variation is that the write-barrier in this technique appears to be expensive for most general- purpose languages. Pure functional and languages with few side effects are not effected. The reason for this is that this write-barrier has to catch all updates and the garbage collector has to ensure that all updates have been propagted when the flip occurs.

2.6.4 Hardware Assistance

Besides offering higher performance and better worst-case latencies than software-based dynamic memory management techniques, hardware-assisted real-time garbage collection also offers more efficient utilization of the memory, both in the average and worse case. Nilsen [13] have described several feasible alter-native real-time garbage collection techniques.

The problem with this is that it is ha ware dependent and can be quite hard to upgrade or modify to fit different stems and special programs. As an example can Linns [7] hardware garbage collector serve. This garbage collector is made as an ASIC. Changes to it is at least costly with respect to fabrication costs. Another problem with Nilsens approach is that he modified the GCC compiler which means that it does not run on standard off the shelf software.

2.5 Choosing Technique

When it comes to choosing the right and be factor is the functions and systems it shall serve. If it is a interactive system then a stop-the-world garbage collector is p case would make simple copying garbage collectors attractive. The other techniques are somewhat interesting but they have to be expanded from their simple form to cope with fragmentation.

With a stop-the-world garbage collector you wait until all memory has been used and when it has all, processes are halted until the garbage collector has finished its garbage cycle.

But since it is hard real-time embedded s stems that is interesting there is only two techniques that is feasible. That is reference counting and copying garbage collectors. Mark and sweep collectors are out of the race since they have disadvantages from both the others. Examples of these are fragmentation, inability to reclaim garbage direct, read /write barrier. This and the cost of the sweep make them deficient and incongruous.
Another thought has to be given to how the garbage collection should be handled if the system is multi-threaded. With multi-threading in a system, the complexity of the garbage collection can incase significantly. Especially if a tracing garbage collector is used. This complexity has to be dealt with since many languages, like Java, has threads as a

prerequisite for serious development. Siebert [22] has presented a software proposal, similar to Nilsens [13] hardware proposal, using synchronization pints to solve this.

The alleged major problem with efficiency in reference counting is not a major problem since it is totally predictable and if it should produce to much overhead then static analysis can cut some of that overhead. Furthermore, when tracing garbage collectors are made incremental they have to use a reader write barrier that make them less efficient With a barrier they also have to check when something happens to a pointer.

When it comes to making tracing algorithms generational, it is a nice idea but it works poor with real-time requirements. This due to the problem that it can not halt the program to recover when a generation gets full.

CHAPTER 3

Fragmentation /Allocation

When a garbage collector with fragmentation issues is selected this problem has to be addressed. This is such a big problem that is deserves its own chapter. Herein will the concept of fragmentation be explained and some different allocation techniques on how it can be dealt with. An important note is that there almost always will exist programs that work bad with even the best of techniques. This will be a brief introduction to the fragmentation/ allocation problem, for a more thorough presentation see Wilson et all[2].

In this chapter overheads in the blocks are omitted unless otherwise specific ally stated.

Following techniques are well known allocation algorithms which deal with fragmentation in different ways.

3.1 Fragmentation: A Quick Guide

External fragmentation occurs when objects of different sizes are allocated and deallocated on the heap. The deallocated objects will leave holes in the continuous memory space. Holes that might not be sufficiently big next time there is an allocation. Since the holes might not be sufficiently big, allocation can fail even if there is enough memory free in total.

Internal fragmentation occurs when objects are allocated in blocks with present size and the object is smaller than the block. The total amount of internal fragmentation is the size difference between the block size and it inherent objects size in all allocated blocks combined.

3.2 Allocator Policy Issues

The problem the allocator must address is fragmentation. This can be done by using different kinds of strategies, placement policies and spitting and cope with this problem.
The fact that reference counting is incremental by nature and that it reclaims garbage almost instantly compensate for the problem of fragmentation and efficiency. Although, these two problems still has to be addressed in the solution and implementation.
In retrospect, a copying garbage collector for timber seams to be quite feasible. This since when threads sleep in Timber their stack is empty which should make it easy to collect

the few objects that live on the heap during this sleep. How this works in reality with the reduced memory, compared to reference counting, and other aspects will be illustrated by Martin Kero in his master thesis. alescing. Among these is the placement choice the main technique for the allocator to control fragmentation. Placement choice is simply the choosing of where in free memory to put a requested block. Despite potential restrictions on an allocator, the allocator has a huge freedom of action. It can place a block anywhere it wants in the assigned memo, as long as there are a sufficiently large amount of memory available. An allocator algorithm therefore should be considered as the mechanism that implements a placement policy, which in him is motivated by a strategy to reduce or control fragmentation.

Which allocator and policy to use can b hard to decide, therefore one could use these issues to get a better idea of ho the allocator should work. When this is done the different algorithms can be compared to see which of them that are appropriate.

- Patterns of Memory reuse. Are recently free blocks reused in preference to older ones? Are free blocks in an area of memory preferentially reused for objects of the same size and type as the live objects nearby? Are free blocks in some areas reused in preference to free blocks in other areas?
- Splitting and Coalescing. Are large free blocks split to meet allocation requests? Are adjacent free blocks merged into larger areas at all? Are all adjacent free areas coalesced, or are there restrictions on when coalescing can be done because it simplifies the implementation? Is coalescing always done when it is possible or is it delayed to avoid needless merging and splitting over short periods of time?
- Fits. When a block of a particular size is reused, are blocks of about the same size used preferentially or blocks of different sizes?
- Splitting thresholds. When a block larger than the request is used, is it split up or does the unused part introduce internal fragmentation? Is the remainder made available for future allocations? If internal fragmentation is introduced, is it because of simplicity or as part of a policy to trade internal fragmentation for reduced non external fragmentation?

The answers to these questions affect the fragmentation of the allocator. The questions or issues above could be en as policies, even if the reason for a particular choice is to make the implementation of the allocator faster or simpler.

3.3 Sequential Fit Algorithm

Usually sequential fit algorithms implementations consists of a doubly linked linear or circularly linked list of all blocks of free memory.Thease lists often alescing. Among these is the placement choice the main technique for the allocator to control fragmentation. Placement choice is simply the choosing of where in free memory to put a requested block. Despite potential restrictions on an allocator, the allocator has a huge freedom of action. It can place a block anywhere it wants in the assigned memo, as long as there are a sufficiently large amount of memory available. An allocator algorithm therefore should be considered as the mechanism that implements a placement policy, which in him is motivated by a strategy to reduce or control fragmentation.

Which allocator and policy to use can b hard to decide, therefore one could use these issues to get a better idea of ho the allocator should work. When this is done the different algorithms can be compared to see which of them that are appropriate.

- Patterns of Memory reuse. Are recently free blocks reused in preference to older ones? Are free blocks in an area of memory preferentially reused for objects of the same size and type as the live objects nearby? Are free blocks in some areas reused in preference to free blocks in other areas?
- Splitting and Coalescing. Are large free blocks split to meet allocation requests? Are adjacent free blocks merged into larger areas at all? Are all adjacent free areas coalesced, or are there restrictions on when coalescing can be done because it simplifies the implementation? Is coalescing always done when it is possible or is it delayed to avoid needless merging and splitting over short periods of time?
- Fits. When a block of a particular size is reused, are blocks of about the same size used preferentially or blocks of different sizes?
- Splitting thresholds. When a block larger than the request is used, is it split up or does the unused part introduce internal fragmentation? Is the remainder made available for future allocations? If internal fragmentation is introduced, is it because of simplicity or as part of a policy to trade internal fragmentation for reduced non external fragmentation?

The answers to these questions affect the fragmentation of the allocator. The questions or issues above could be en as policies, even if the reason for a particular choice is to make the implementation of the allocator faster or simpler.

Use Knuth's boundary tag technique to support coalescing of all adjacent free areas. For more information about this see. Knuth [11]. To manage the list a FIFO, LIFO or address order is usually used. From this list allocation can be done in three basic ways, these are first fit, next fit and best fit.

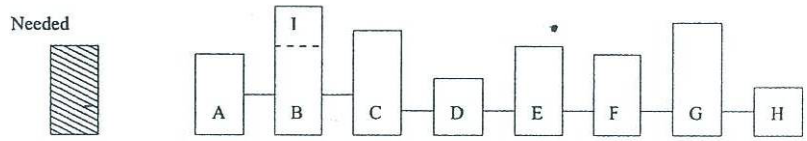


Figure 3.1: Sequential fit algorithms

- First Fit. This allocator searches the list f free blocks from the beginning and uses the first one big enough. If the selected block is larger than-necessary then it divides the block so t the request IS full-filed and the reminder is placed back on the free list . This tend to split large objects in the beginning of the list and adding the small reminders at the end. Thus making poor use of the small objects. In figure 3.1 this means that block B will be used and that the reminder I is placed at the back of the list. When the next allocation occurs the algorithm will begin to examine, if block A is big enough.
- Next Fit. This is an evolved version f first fit, commonly seen as an optimization. It uses a roving pointer for allocation. This pointer makes sure that the allocator does not start over is an allocation. The idea is that since algorithm starts at the place were it found of small unusable blocks will occur in 0 means that block B will be used and before block C. So that when the next will begin to examine if block C is big allocation. This pointer makes at the beginning each time there the pointer makes sure that the d the last block no accumulation of small unusable blocks will occur in one part of the list. In figure 3.1 this means that block B will be used and the roving pointer will be placed before block C. So that when the next allocation occurs this algorithm will begin to examine if block C is big enough.

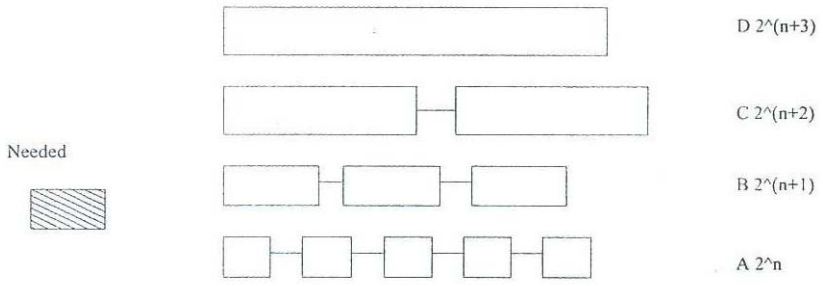
- Best Fit. This algorithm searches the Ii t for the smallest available block large enough to satisfy the requirement. Thus trying to minimize the amount of wasted space by ensuring at the fragments are as small as possible. This is an comprehensive search if a perfect fit is not found. Then it is may stop depending on ho it is implemented. In figure 3.1 this means that the list will be search d from block A to block E, this since block E is a perfect fit. If the bloc E had not existed it would have examined the hole list and then chose block C since that is the best fit.

Due to the search, this technique is considered inefficient, and it can be so in terms of speed but in terms of frequentation it can surpass the others.
Furthermore, when combined with a segregated fit algorithm the search is considerably shortened which should make this technique interesting in some situations. More about the segregated fit algorithm in the next section.

3.4 Segregated Free Lists

This allocation technique, one of the simple , uses a number of lists were each list holds free blocks of different preset size The size of these blocks are often a power of 2 apart (e.g., 4 words, 8 words 16 words and so on). When an allocation occurs the objects size will be rounded off upwards towards the nearest boundary, in this example the next higher power of 2, and a block of that size will be allocated. Depending on their size blocks and objects are said to belong to different size classes.

The reason to have different free lists is to get away from the most naive solution where the preset block size is the s me as the biggest possible object, including some overhead. If there only s one list then the internal fragmentation would be overwhelming. This since there normally are a few data structures that are big and a lot of variables hat are small in a program.



- Simple Segregated Storage. When there are not enough small blocks this variant will split large blocks so that e allocation request can be satisfied. Although it will not merge small blocks back into bigger ones. The split is usually done by asking the operating system for help. The operating system then divides the memory to two equal sized blocks that are placed on their free list so that the request can be served. In figure 32 an allocation call has been made. The blocks in list B 2(n+1) has the closest upper bound and therefore a block from that list will be allocated.
- Segregated Fit Algorithms. In this variant different blocks in the same size class do not necessarily have the same size. They are only about thesame size, how big difference thats all wed depends on how far apart the different size classes are. This is a more relaxed policy that can work well when combined with for example best fit.

3.5 Linked list with Indirection

In this variant there is only one list of free blocks and all the blocks have the same size. This size is preset and can be dived from static analysis in the compiler or manual analysis so that it is op mal, or at least acceptable. The block size is optimal in the sense that the internal fragmentation is minimal. During these analysises consideration is take to how big objects are and how often each different object size is allocated. The derived block size is most certain smaller than the bigger objects, so special cases exists where the block size is equal to the.l2iggest object. The during allocation, it is up to the allocator to provide blocks and divide the objects that are bigger than the block size to make them fit in the fewest number of blocks.

To simplify the example in figure 3.4 the normal overhead is assumed to be only one word. As seen in the example t e object is bigger than the block size. Due to that, the allocator have to divide the object into as many blocks as necessary, in this-example three blocks.

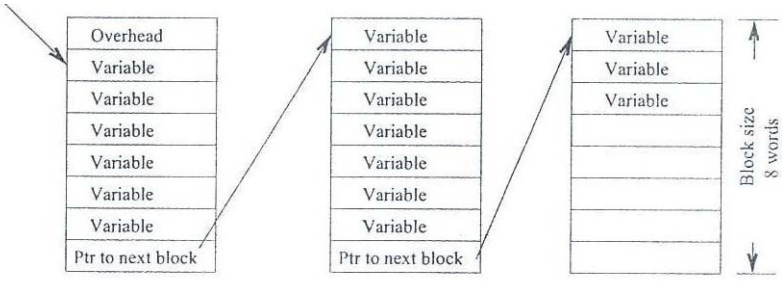


Figure 3.3: Indirection type 1. A 16 words object in three 8 words blocks.

With some extra complexity the number of indirections can be reduced. This is done by putting all the pointers to other blocks within the object in the first block. The point of this is so that when big objects that uses a lot of blocks are accessed it should be easier and faster to access data towards the end of the object. Furthermore, it creates a more even cost to access data from the object.

Using the same numbers as in previous e ample, this variation has one less indirection, see figure 3.5. As can be seen in is example the maximum object size with only one indirection is 8WOTds*7blo ks = 56words since the first block only can hold pointers to seven new block. After that another indirection from the second line of blocks have to be issued.

A third variant of this technique is to do the division of big objects statically during compile time. This is done by object transformations. All objects bigger than the specified block size will be transformed into two or more objects. One

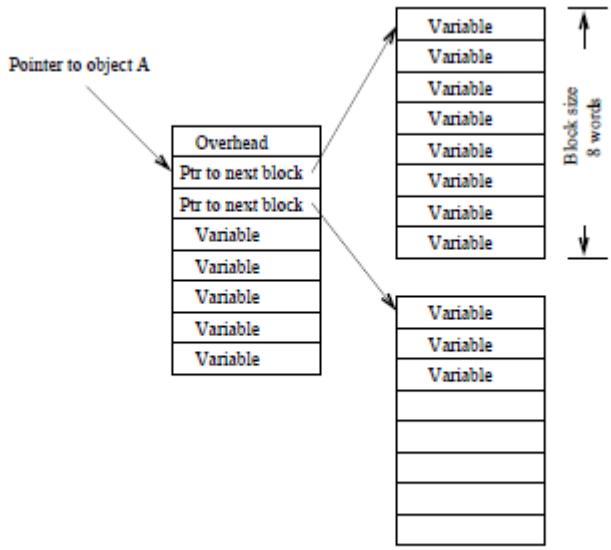


Figure 3.5: Indirection type II. A 16 words object in three 8 words blocks.

Could argue if its variation should have its own section since in the compiler who arranges all indirections and makes these transparent to the allocator.

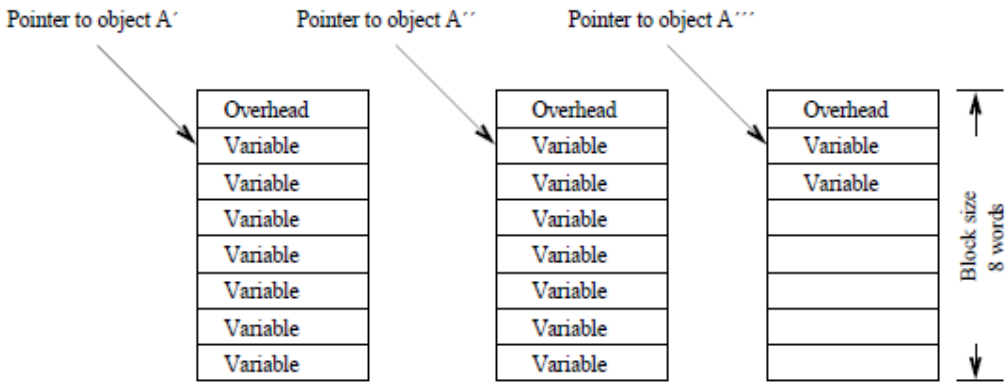


Figure 3.6: Indirection type III. A 16 words object in three 8 words blocks.

As can be seen in figure3.6, object A has been transformed into the three objects A',A'',A'''. The internal fragmentation is the same for all the three variants. The only difference is how and when the indirection is done.

The gain of this is that except from saving run-time, these objects can be reclaimed before the others, i.e. a part of the original object can live forever and therefore be allocated forever. With this variant this allocation holds only one block in contrast to the previous were several blocks can be allocated. This example is a bit extreme but still makes a point.

3.6 Choosing Technique

Firstly it has to be said that there are a couple of advanced versions of these fundamental techniques , like Chung and Moon[3] and Rezaei and Cytron[17].So there are possibilities that, I the techniques described in this report just is not good enough or have other flaws, a perfect fir for a specific system or program already exists.

The reason the sequential fit section in this report is that although these techniques seems pretty crude they are not always dismissible just by looking at them. According to John stone and Wlison[10] these techniques have a smaller true fragmentation than the others. This is when observed and things like that are considered. So they can not be dismissed just like that in systems where external fragmentation is of less concern. In real-time system it is. So therefore are they dismissed right away.

External fragmentation is very unpredictable and therefore totally unacceptable in real-time systems. This leaves the techniques that suffer from internal fragmentation. The transition between external to internal fragmentation is easy but it comes at some costs.

The cost is that of complexity and memory utilization is different. But this cost is easy to pay for the reward in a predictable system.

As said before the easiest way to do the transition is to use a preset block size. Since most languages allow different objects of different sizes internal fragmentation occurs when a object is placed in a block that is bigger than the object. The most naïve implementation us to have a block size as big as the biggest object. This could work out in some cases but for most of the time it will waste a lot of memory. Therefore, a smarter technique has to be used which one is of course dependent on language and compiler specific issues.

CHAPTER 4

PROBLEM STATEMENT

We argue that a programming language meant for writing applications (i.e. end-user applications, as opposed to low-level systems code) must have automatic memory management in order for such applications to be maintainable and modular.

Uniform Reference Semantics

We have already argued elsewhere that uniform reference semantics is a prerequisite for modular and maintainable code. If you don't know what that means, you should first look at that text and come back only when you are convinced that that argument is correct.

Multiple References To Objects

The key argument to needing automatic memory management is that it is impossible, or at least very hard, for a program to know when an object can be safely deleted, or equivalently, when an object is about to have its last reference removed. Let us use the same example that we use elsewhere. Suppose a program contains two containers, one containing all the staff of the company, and another containing all the people with parking permits. For that program to be modular, the part that manipulates parking permits must be independent of the part that manipulates staff members. Suppose someone quits his or her job, thus no longer begin a staff member. That person is then removed from the corresponding container. Should we delete the object representing the person as well, or should we just remove that object from the container? If that person is also in some other container, then we should not delete the object, but if that person is only in the container representing staff members, we should delete the object. For instance, if the person also has a parking permit, then the object is also referenced from the container of holders of parking permits, and we absolutely must not delete the object. Doing so anyway will either result in a core dump or, worse, in some mysterious bug in which a person without a parking permit still is in the container for parking permits. But making such a decision requires the part of the program (module) that manages staff members to know about all other modules, in particular about parking permits. Needless to say, such global knowledge would completely ruin modularity and any hope for producing a maintainable program.

Complication Of Code Dues To Manual Memory Management

Some code is much more complicated than it ought to be because of lack of automatic memory management. Let us suppse we have som C function concat that concatenates two strings by allocating a new string the size of the two arguments, and copying the characters to the newly allocated space. Now suppose we want to concatenate **three** strings , x, y, and z. We could do something like this:

```
w = concat(concat(x, y), z);
```

However, this code has a **memory leak**. The inner call to concat allocates memory that is never deallocated. A simple idea like this must be expressed this way instead to avoid memory leaks:

```
temp = concat(x, y);
w = concat(temp, z)
free(temp);
```

How does automatic memory management solve the problem?

Automatic memory management solves the problem by detecting objects and groups of objects that cannot possibly be referenced by a program, and deleting them. With automatic memory management, we can safely remove the (reference to the) object contained in the container of staff members without deleting the object itself. Should it be the case that doing so will make that object inaccessible, for instance if it is contained in no other container, then the automatic memory management system (the garbage collector) will eventually detect it and delete it. Conversely, if the object is still referenced from a container (that itself is accessible), then the garbage collector will preserve the object intact.

How does garbage collection work?

To do its work, the garbage collector needs to know which objects cannot be accessed. It does so by directly applying the definition of `accessible':

1. Any object referenced from a machine register is accessible.
2. Any object referenced from a global variable is accessible.
3. Any object referenced from the stack is accessible.
4. Any object referenced from an accessible object is accessible.
5. Any object not accessible according to the previous definition can be deleted.

Usually, the garbage collector traces all the accessible objects starting with the registers, the global variables, and the stack. While tracing these objects, it inserts them in a set (for instance by marking them). When all accessible objects are traced, the remaining ones (the ones not members of the set) are deleted.

But isn't garbage collection slow?

Not really. But it is hard to evaluate the exact cost of it, since not having it means writing your program differently. If not having a garbage collector means lots of duplications of objects, then having it is probably cheaper. Also, if not having a garbage collector means having a buggy program, you can argue that the cost of not having one is infinite.

> To get an estimation, you may consider that the execution time of a typical garbage collector is less than 15% of total execution time of the program. But without the garbage collector, at least part of that time would have to be devoted to explicit calls to `free'.

So it isn't slow, but doesn't it have these unacceptable pauses?

Some garbage collectors do, especially old ones. For batch programs such pauses are not a problem, and garbage collectors of that sort are usually very efficient.

> For interactive programs, pauses can be annoying. More recent garbage collectors, such as generational collectors, only scan a small part of all objects during each collection, making the pauses hard or impossible to detect.
>
> Incremental collectors go one step further and interleave their execution with that of the application so that no pauses remain, only a minor slowdown of

the application. Such collectors can even be used in real-time systems with hard requirements on response times.

So can't we use reference counters?

A common trick, often used in languages such as C or C++ with no automatic memory management is to use reference counters. The idea is that each object contains an additional field that indicates how many other objects refer to it. Each time another reference is created (for instance when the object is inserted into another container), the reference counter is incremented. Each time a reference to the object is lost (for instance when the object is removed from a container), the reference counter is decremented and checked. If its value is zero, then no references to the object remain and the object can be removed.

Reference counters can be much more costly than automatic memory management, as the reference counter needs to be updated fairly often, usually much more often that a garbage collector would scan the object.

Worse, reference counters fail to detect some inaccessible objects, thus potentially creating memory leaks. The reason is the presence of cycles of references. Objects in such a cycle will have their reference counters all greater than zero, despite the possibility of none of the objects being accessible. All of them are accessible from each other, but non of them from the registers, from the global variables, or from the stack, neither directly nor indirectly. By the definition of accessibility, they are thus not accessible. Despite that, their reference counters are greater than zero, making it impossible to delete them.

The only situation in which reference counters work, is when one can guarantee that no object participate in a cycle, which for some applications is possible. But even then, reference counters represent a greater cost than that of automatic memory management.

Languages with automatic memory management

From the explanation above, we must conclude that automatic memory management is a necessity for programming applications. Many languages that were created for that

purpose therefore directly support automatic memory management, in particular Simula, Eiffel, Lisp, Scheme, Java, and Smalltalk.

What does it mean that the language `supports' automatic memory management? First, it means that all implementations are required to have it. But it also means that the language is designed so as to make it easy for a garbage collector to precisely detect live (accessible) objects.

This is not the right forum to go into a deep discussion as to how languages might make it easy for the garbage collector. For those who are interested, there is a large body of literature on automatic memory management. It is also a very active research topic.

What if the language does not have automatic memory management?

Some very popular languages, in particular C and C++ (and Ada?) do not support automatic memory management. The best solution is of course to avoid these languages for writing applications. Sometimes, however, that is not an option.

In those cases, for C and C++, the best solution is to use a so called conservative garbage collector. A conservative garbage collector does not need the support from the compiler or the language in order to do its job. It is called `conservative' because it may erroneously declare an object as `live' even though in fact it is not, whereas it can never (provided the programmer follow some elementary precautions such as respecting the language definition) erroneously declare an accessible object as inaccessible.

CHAPTER - 5

Real time systems and garbage collection

Real-time (RT) application development distinguishes itself from general-purpose application development by imposing time restrictions on parts of the runtime behavior. Such restrictions are typically placed on sections of the application such as an interrupt handler, where the code responding to the interrupt must complete its work in a given time period. When hard RT systems, such as heart monitors or defense systems, miss these deadlines, it's considered a catastrophic failure of the entire system. In soft RT systems, missed deadlines can have adverse effects -- such as a GUI not displaying all results of a stream it's monitoring -- but don't constitute a system failure.

In Java applications, the Java Virtual Machine (JVM) is responsible for optimizing the runtime behavior, managing the object heap, and interfacing with the operating system and hardware. Although this management layer between the language and the platform eases software development, it introduces a certain amount of overhead into programs. One such area is GC, which typically causes nondeterministic pauses in the application. Both the frequency and length of the pauses are unpredictable, making the Java language traditionally unsuitable for RT application development. Some existing solutions based on the Real-time Specification for Java (RTSJ) let developers side step Java technology's nondeterministic aspects but require them to change their existing programming model.

Metronome is a deterministic garbage collector that offers bounded low pause times and specified application utilization for standard Java applications. The reduced bounded pause times result from an incremental approach to collection and careful engineering decisions that include fundamental changes to the VM. Utilization is the percentage of time in a particular time window that the application is permitted to run, with the remainder being devoted to GC. Metronome lets users specify the level of utilization an application receives. Combined with the RTSJ, Metronome enables developers to build software that is both deterministic with low pause times and pause-free when timing windows are critically small. This article explains the

limitations of traditional GC for RT applications, details Metronome's approach, and presents tools and guidance for developing hard RT applications with Metronome.

5.1 Traditional GC

Traditional GC implementations use a stop-the-world (STW) approach to recovering heap memory. An application runs until the heap is exhausted of free memory, at which point the GC stops all application code, performs a garbage collect, and then lets the application continue.

Figure 1 illustrates traditional STW pauses for GC activity that are typically unpredictable in both frequency and duration. Traditional GC is nondeterministic because of the amount of effort required to recover memory depends on the total amount and size of objects that the application uses, the interconnections between these objects, and the level of effort required to free enough heap memory to satisfy future allocations.

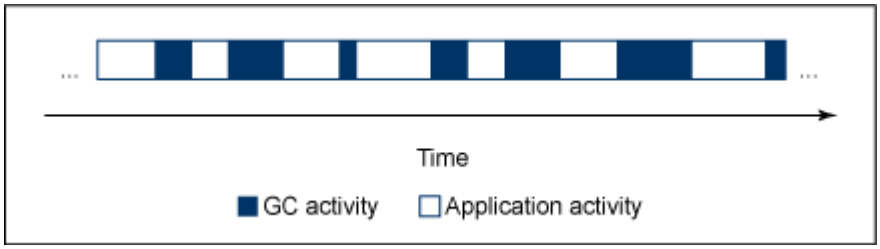


Figure5.1. Traditional GC pauses

Why traditional GC is nondeterministic

You can understand why GC times are unbounded and unpredictable by examining a GC's basic components. A GC pause usually consists of two distinct phases: the mark and sweep phases. Although many implementations and approaches can combine or modify the meanings of these phases, or enhance GC through other means (such as compaction to reduce fragmentation within the heap), or make certain phases operate concurrently with the running application, these two concepts are the technical baselines for traditional GC.

The mark phase is responsible for tracing through all objects visible to the application and marking them as live to prevent them from having their storage reclaimed. This

tracing starts with the root set, which consists of internal structures such as thread stacks and global references to objects. It then traverses the chain of references until all (directly or indirectly) reachable objects from the root set are marked. Objects that are unmarked at the end of the mark phase are unreachable by the application (dead) because there's no path from the root set through any series of references to find them. The mark phase's length is unpredictable because the number of live objects in an application at any particular time and the cost of traversing all references to find all live objects in the system can't be predicted. An oracle in a consistently behaving system could predict time requirements based on previous timing characteristics, but the accuracy of these predictions would be an additional source of non determinism.

The sweep phase is responsible for examining the heap after marking has completed and reclaiming the dead objects' storage back into the free store for heap, making that storage available for allocation. As with the mark phase, the cost of sweeping dead objects back into the free memory pool can't be completely predicted. Although the number and size of live objects in the system can be derived from the mark phase, both their position within the heap and their suitability for the free memory pool can require an unpredictable level of effort to analyze.

5.2 Traditional GC suitability for RT applications

RT applications must be able to respond to real-world stimuli within deterministic time intervals. A traditional GC can't meet this requirement because the application must halt for the GC to reclaim any unused memory. The time taken for reclamation is unbounded and subject to fluctuations. Furthermore, the time when the GC will interrupt the application is traditionally unpredictable. The time during which the application is halted is referred to as pause time because application progress is paused for the GC to reclaim free space. Low pause times are a requirement for RT applications because they usually represent the upper timing bound for application responsiveness.

CHAPTER - 6

Java garbage collection for real time systems

The program specifications and hardware constraints of embedded systems result in some unique problems when it comes to garbage collection. Therefore, depending on the real-time-systems application, different garbage-collection algorithms are appropriate.

The efficiency of a garbage-collection algorithm depends on numerous factors -- the hardware of the targeted platform (its CPU type, amount of memory available, and availability of memory-management hardware support), the type of run-time system (compiled or interpreted), and the level of support it provides for memory allocation (supervised or unsupervised). Finally, the application itself and its memory usage patterns (memory cells connectivity, relative size, and life span of allocated objects) are also relevant.

When these factors have been taken into consideration, suitable garbage-collection algorithms should be reviewed in terms of their processing costs, the space overhead they impose (both on the allocated data and on the code size), and the performance degradation they incur as the heap fills up.

In this article, I'll define and describe garbage collection, discuss the issues involved in choosing a garbage-collection scheme for real-time systems, and describe the garbage collector used by Kaffe, a freely available Java VM we have experimented with.

6.1 Definitions

Garbage collection relies on the observation that a previously allocated memory location, which is no longer referenced by any pointers, is no longer accessible to the application and is therefore reclaimable. The role of a garbage collector is to identify and recycle these inaccessible allocated memory chunks. There are two families of garbage-collecting techniques -- reference counting and tracing.

Reference counting keeps track of the number of references to a memory cell. The cell is considered reclaimable whenever this number becomes zero. The compiler or the run-time system is responsible for adjusting the reference count every time a pointer is

updated or copied. The main advantage of reference counting is that the computational overhead is distributed throughout the execution of the program, which ensures a smooth response time. Reclaiming an isolated cell doesn't imply accessing other cells, and there are optimizations that reduce the cost of deleting the last reference to an entire group of cells. The major weakness of reference counting is its inability to reclaim cyclic structures such as doubly linked lists or trees featuring leaf nodes with pointers back to their roots.

Reference counting can be optimized to reduce the number of times reference count fields have to be updated and to reduce the size of the reference count field. It can also be modified to detect pointers that are part of a cycle or can even be combined with a tracing algorithm to reclaim cyclic data.

Reachable (live) memory locations are those that are directly or indirectly pointed to by a primordial set of pointers called the "roots." Tracing algorithms start from the roots and examine each allocated memory cell's connectivity.

There are two kinds of tracing algorithms -- mark-and-sweep and copying. Mark-and-sweep marks active memory cells (those that are reachable through pointer reference, starting with the roots) whenever the amount of free memory goes below a certain threshold, then sweeps through the allocated cells. When sweeping is complete, unmarked cells are returned to the available memory pool. The advantage of this approach is its ability to reclaim cyclic data structures. If implemented as I've just described, however, it may not be acceptable for an application with some real-time constraints.

Copying algorithms use a heap divided into halves called "semispaces." One semispace (the "current space") is used for memory allocation. When garbage collection occurs, active memory cells are traced from the current space and copied into the second space. When done, the second space contains only live cells (nongarbage) and becomes the new active space. The cells get automatically compacted as a result of this process. Unreachable cells (garbage) are not copied and are thus naturally reclaimed. The copying algorithm ensures fast allocation of new cells and reduces fragmentation. On the other hand, it halves the memory available to the application and touches all the living memory cells during the copy algorithm, which may lead to a considerable number of page faults on a system using pagination.

Tracing garbage collectors can be either accurate/exact (able to distinguish pointer from nonpointer data) or conservative (consider every word encountered as a potential pointer). The type of tracing garbage collector you use -- exact or conservative -- affects the way

pointers are identified within the location where roots can be found (registers, stack, static areas) and the way the garbage collector scans cells looking for pointers to other cells.

A conservative garbage collector needs to examine every memory word as a potential pointer, and may misidentify a word as a pointer, thus retaining memory cells that are actually garbage. Since a fully conservative garbage collector can't risk moving memory blocks around, wrongly updating memory references it thinks point to them, it cannot be a fully compacting collector.

Conservative collectors can be altered to be mostly copying: Objects that are solely accessible from other heap-allocated object are copied. They are deployed on systems where no cooperation from the compiler or the run-time system is expected to help the collector tell whether a word is a pointer or not.

6.2 Garbage Collection and Java

The Java run time relies on garbage collection to automatically identify and reclaim objects no longer in use. Garbage collection doesn't alter the behavior of an application written in Java, with the exception of classes that declare a finalizer method. Finalizers are executed right before objects found to be garbage are discarded, giving you a chance to explicitly release nonmemory resources the class uses. Tricky finalizers may resuscitate an object by installing a new reference to it, but finalizers are executed only once. This guarantees that objects may be brought back to life only once. Beginning with JDK 1.1, data belonging to classes is also garbage collected, allowing unused classes to be automatically discarded (this happens whenever their class loader gets recycled). System classes are never garbage collected.

The initial set of roots of a Java run time are the live thread objects, system classes (for instance java.lang.String), local variables and operands on the JVM stack, references from the JNI, pending exceptions, and references from the native stack. Stemming from these roots, all three types of Java entities feature reachable components that will be examined by the collector. Objects have instance fields and classes that are reclaimable, arrays have element objects and classes that can be garbage collected, and classes feature a set of components that can be recycled: super class, interfaces, linked classes, string literals, static fields, class loader, signers, and protection domains.

Up to JDK 1.1.4, Sun used a partially conservative compacting mark-and-sweep algorithm, which makes a conservative assumption when the native stack is scanned, but allows the Java heap and stack to be scanned in a nonconservative way. The algorithm compacts the heap, substantially reducing its fragmentation. Sun's implementation relies

on an implementation detail to keep the cost of objects' relocation low. The reference to an object (by which an object is known to exist to other objects and the VM) is implemented as a nonmoving handle pointing to the actual object's data. Once the actual object's relocation is performed, simply updating its handle allows the object to be accessed at its new location.

The garbage-collection code is executed in a separate thread when the system runs out of memory or when System.gc is explicitly invoked. The collector may also run asynchronously when it detects that the system is idle, but this may require some support from the native thread package. In any cases, execution of the garbage collector halts all other application threads and scans the entire heap.

6.3 Garbage Collection for Embedded Java

To adapt Java garbage collection to embedded systems, you must consider:

The nature of the run-time environment.

The hardware characteristics and limitations of the embedded platform.

The expected performances (acceptable general overhead and maximum pause times).

The way in which critical memory situations need be handled.

The run-time environment will mainly influence the way roots are identified. An interpreted or semicompiled (JIT) environment can make references clearly identifiable so the Java heap and the Java stack can be scanned nonconservatively. On the other hand, the precompiled approach (like that being developed at Cygnus) needs to rely on conservative scanning, unless additional information is made available to the run time by a cooperative compiler.

Memory -- both RAM and ROM -- is a scarce resource. Memory constraints usually rule out all copying and generational algorithms and prevent sophisticated solutions like adaptive collectors (which dynamically change the way they garbage collect the memory to achieve greater efficiency) or cooperative collectors (which feature several algorithms in one collector, to get the best of several worlds) because of the amount of ROM required for their implementations.

Additionally, reference counting requires the count field of an allocated object to be roughly the same size as a pointer or some other value that can represent all valid addresses of the system. This field has to remain tied to the object. On the other hand, information tags required by a mark-and-sweep range from a minimum of one bit to several bits for the implementation of three colors algorithms (which are based on an abstraction introduced by Dijkstra, attributing different colors to memory cells depending

on the need for the collector to visit, reconsider, or discard them) or pointer-reversal marking algorithms (a time and space bound algorithm that stores in the currently examined memory cells the value of its parent pointer).

Most embedded systems don't provide swapped virtual memory, which implies that accessing a random portion of the memory has a fixed cost. This diminishes the impact of the mark-sweep and the space-costly copying collectors that visit all live cells before reclaiming the dead ones. On the other hand, this also means the heap can't be expanded. When reaching a low-water mark situation, the collector can't count on tapping some virtual memory as a temporary solution.

An embedded application running Java has a strong chance of being interactively operated by users. This demands a fast, nondisruptive garbage collector that also provides predictable performance for future memory allocation. Periodically performing partial garbage collection must be sufficient to prevent the need to run the garbage collector when a cell is allocated. However, no noticeable slow down should occur in the extreme case that the collector is forced to run.

You can enhance predictability by improving heap fragmentation so that the low-level operation of reserving memory doesn't have to scan the heap trying to find sufficient free memory for an indeterminate amount of time. It should also be guaranteed to succeed if enough free memory is known to be present. It is unacceptable for an embedded system to have a memory allocation failure because the fragmentation of the heap prevents the allocator from finding a memory block of a sufficient size.

Fragmentation can be fought by compacting the heap. While compacting the heap, cells can be moved either by using a technique called "sliding," where the cells' allocations are retained, or by using a technique called "linearizing," where a cell pointed to by another cell is moved to a close adjacent position. In addition to the copying algorithm, which compacts the heap as a side effect of copying a live cell from one semispace to another, there are several compacting mark-sweep algorithms (also called "mark-compact") with complexities ranging from O(M) to O(M log M), where M is the size of the heap. Conservative scanning imposes restrictions on objects that can be moved.

6.4 Nondisruptive Garbage Collection

To be able to provide nondisruptive garbage collection, it is necessary to use an incremental algorithm that runs for a fixed amount of time and ensures that further memory allocation requests won't fail. Unfortunately, real-time algorithms usually rely on additional processors or huge amounts of memory so that they can run copying type

collectors. The Wallace-Runciman algorithm combines mark-during-sweep where portions of the heap are marked and then swept as the mark phase works on the next region, and stack collect, a modified mark-sweep where states of the marking phase are pushed on a stack. Stack-collect avoids mark-during-sweep worst case execution time behavior, which is quadratic to the size of the heap. This algorithm has a small three bits per cell overhead, can operate with small stack depth, and has bounded execution time. Unfortunately, its execution time accounted for 30 to 45 percent of an application run time in early implementations, targeting a functional language and assuming a heap exclusively made of binary cells.

Baker's incremental copying algorithm is a real-time collector that doesn't really fit in a embedded environment. It requires at least twice the minimum amount of memory to be implemented. Wilson and Johnstone devised an incremental noncopying implicit reclamation collector that meets real-time constraints. It uses a write barrier to mark pointers to nonexamined objects stored in cells that have already been examined so they are later reexamined by the collector. This is combined with a noncopying implicit reclamation strategy where unreached objects don't need to be traversed to be reclaimed. The main cost resides in the necessity of maintaining data structures used to keep track of reachable cells in order to deduce unreachable (garbage) objects. The algorithm can be altered in order to fight the fragmentation it naturally induces and requires a strong cooperation from the run-time system side (or the compiler) to implement the write barriers efficiently.

6.5 The Three-Colors model

The three-colors model (first introduced by Dijkstra) is useful to describe incremental or nonincremental tracing garbage collectors. It assigns a color to a heap cell depending on the degree of completion of its collection. At the start of collection, all cells are white. If the cell and its immediate descendants have been visited, it is colored black and won't be reclaimed. A cell entirely visited with the exception of its immediate descendants is colored gray. If the mutator can run in parallel with the collector and changes any pointer on a black cell, it has to change its color back to gray. Consequently, the collector must reconsider a gray cell since its collection was incomplete or changed by the action of a concurrently running mutator.

Under the three-colors model, a collector runs until all reachable cells are blackened. White (unvisited) cells are garbage and hence reclaimable.

The source code for the virtual machine found under kaffevm/ features two files that are relevant to the garbage collection.

- gc-mem.c implements the GC-aware memory allocator and some low-level GC operations.
- gc-incremental.c implements the conservative collector.

Kaffe distinguishes between several allocation needs. On the one hand, it has to deal with Java objects (classes, arrays, and class instances). On the other hand, it has to allocate memory for different internal data structures that are required by the VM and its components (like the JIT) to function properly. All of these pieces of memory need to be garbage collected and/or finalized differently.

For example, the memory allocated for a class instance may contain references to other objects and is scanned (walked) conservatively, but also may or may not require finalization. The actual implementation that entirely and conservatively walks the memory allocated for a class instance isn't the most efficient. It could be optimized by using the object's class information to distinguish, in the object's memory, references to other objects from noncollectable data.

On the other hand, the data structure representing a java.lang.Class object contains a lot of fields. It turns out that only some of them need to be walked (sometimes optionally), and classes are never finalized. Some other data structures, like thread locks or global paths, are never garbage collected. Finally, live threads and native stacks need to be scanned conservatively, because they may contain references objects.

To deal with these requirements, one piece of bookkeeping information Kaffe ties to a garbage-collected chunk of memory is a pointer to a particular GC function set -- a data structure featuring two function pointers. The first one is the collection routine and defines how the allocated chunk of memory should be walked. The second either specifies an object's finalization routine, or is used as a flag to mark the memory chunk's special properties, tagging normal, fixed, or root objects.

This data structure forms the typedef gcFuncs and several elements of that type are statically defined in gc-incremental.c.

When the garbage collector has to operate on a memory chunk, it retrieves its private GC function set and applies the function that is required to walk or finalize it. Table 1 summarizes how some of the memory allocations serving particular purposes are linked to a dedicated GC function set.

6.6 Allocating Garbage Collected Memory

The main function used to allocate garbage collected memory, gc_malloc(), allocates a memory block guaranteed to be big enough to fit the requested size by calling gc_heap_malloc(), which belongs to the low-level memory allocator. It then ties to that block a pointer to an instance of the gcFuncs data structure. For example, to allocate 52 bytes of garbage collected memory that will represent an object's instance that needs to be finalized, you can write void *newObject = gc_malloc (52, &gcFinalizeObject);.

The GC function set is also used to determine whether the allocated memory is a GC root, in which case it is linked to the rootObjects list. For the allocation of a collectable object, gc_malloc() also sets its color to WHITE and inserts the whole garbage-collected page it belongs to in the white list.

The function gc_malloc_fixed() is a short cut to the invocation of gc_malloc() with the gcFixed GC function set as a second argument. There are other functions that can attach or reattach a random piece of memory to the garbage collector. In that case, an indirect root is created and collected separately.

6.7 Garbage Collector and Finalizer Execution

Created during the initialization phase of Kaffe, the conservative collector runs as a separate daemon thread set to run the gcMan() function. At the same time, the finalizer thread is created with a hook to the finalizerMan() function.

6.8 The Garbage-Collection Process

There are several possible invocations of the garbage collector. gcMan(), whose code is periodically executed by the GC thread, is an endless loop. After having made other threads wait for it to complete, it calls the function startGC(). This processes all root objects, sets their color to white, and calls markObject() on them. markObject() first makes sure that the object belongs to garbage-collected memory by calling

gc_heap_isobject(). The object is then turned gray, and the page to which it belongs is moved to the gray list.

startGC() ends by calling walkLiveThreads(), which processes the list of live threads object and calls walkMemory() on each of them. For live threads, the private GC routine is walkThread(), which calls markObject() on private thread info and then applies walkConservative() on the thread's stack, looking for object references to be collected.

gcMan() continues by processing the list of gray-allocated pages that it eventually obtained from the startGC() pass, and calls the function walkBlock() on them.

The white (garbage) list is processed, and objects that need a finalization are marked calling markObject(). Next, potentially new gray objects are processed the same way the gray objects were previously processed.

finishGC() is called to process garbage (white) objects. Objects that need to be finalized are moved to the finalize list. Those that don't require finalization but are white are returned to the local free storage of the page to which they belong, calling gc_heap_free(). The finalizer thread is then awakened. Finally, black objects are moved back to the white list and marked accordingly.

Another way of invoking the garbage collector is by calling invokeGC(), which basically wakes up the GC thread. This is used to implement native functions like System.gc() or Runtime.gc().

6.9 Finalization

The finalizer thread, periodically allowed to run by the GC thread, walks the finalize list and processes entries (which are pages of garbage-collected memory) as follows: The page is first moved to the white list. Then, elements of the page marked to be finalized are individually processed by their own finalizer function.

6.10 Preventive Measures

The heap in an embedded system can't be expanded because of the lack of virtual memory. This makes running out of memory a critical situation that applications should avoid by using preventive measures. Since it's impossible for a Java application to

explicitly free memory (it can only make its best effort to ensure that references to an object are suppressed), it can be worthwhile to encourage references to huge but easily recomputable objects using weak references.

Weak references were introduced in JDK 1.2. They allow references to an object to be maintained without preventing the object from being reclaimed by the collector. Applications may also use the MemoryAdvice interface to be implemented by the RunTime class in JDK 1.2. It defines the getMemoryAdvice method that returns four indicators of the memory usage, as estimated by the run-time environment, from GREEN (everything is fine) to RED (take every conceivable action to discard objects).

In the case of an application running out of memory, an aggressive garbage-collection pass should be performed. The garbage collector must be guaranteed the memory this operation requires. Since some stack and some heap may be necessary, the system needs to monitor the low-water mark, and prevent the application from totally running out of memory before taking action. If no memory can be found, the system throws an OutOfMemoryError error and it's up to the application to selectively choose which parts should be discarded.

6.11 Flexibility Required

Being able to characterize the dynamic-memory-allocation behavior of an application helps in choosing a garbage-collection algorithm. For example, if it appears that the embedded application is known to build no cyclic data structures, then a derived reference-counting algorithm might be appropriate. If allocated objects are known to be fairly similar in sizes, then a noncompacting algorithm might be used.

Instrumentation of memory-allocation behavior, including peak size, object sizes, and life spans statistics will help a developer to select the correct garbage-collection algorithm. Such a monitoring can be performed during development and the test phases by using a special version of the run-time environment. The next step is to assist you by allowing you to plug and tune the garbage collector of your choice.

We're examining these issues at Cygnus Solutions, where we are developing a Java run time and Java extensions for the GCC compiler (see the accompanying text box entitled "The Cygnus GNUPro Compiler for Java"). Being in control of the tools that generate the

code gives us opportunities to ease the work of a garbage collector in many ways. We can precisely perform any relevant static allocation of objects and implement write barriers, even though such implementation may be challenging if we want to support pluggable collectors. An efficient write barrier needs to interact very closely with the collector in use, being aware of either its internal data structures or the object marking in use. They usually feature between 10 to 30 instructions and can't really afford the versatility of using function calls. One solution might be to specify the write barrier as a user-editable piece of code, which can be inserted as necessary by the compiler during the instruction generation phase. Developing our own compiler also gives us a chance to analyze the way finalizers are written, thus detecting actions that resuscitate objects being thrown away. This could be done by analyzing the use of this as an expression right side or method parameter.

CHAPTER - 7

Real-Time garbage Collection in the Sun Java Real-Time System (Java RTS)

Sun Java Real-Time System (Java RTS) is Sun's commercial implementation of the Real-Time Specification for Java (RTSJ), or JSR 1.

Starting with Sun Java RTS 2.0, a new real-time garbage collector (RTGC), based on Henriksson's algorithm, is available. The garbage collector runs as one or more real-time threads (RTTs). These run at a priority that is lower than all instances of NoHeapRealtimeThread (NHRT) and that may be lower than some RTTs as well, so that critical threads may preempt the collector. In this way, critical threads are shielded from the effects of GC.

One of the tuning parameters in the RTGC algorithm is the maximum priority of the garbage collector. This divides the priority range into regions. The NHRTs receive the highest priority and are known as critical tasks. Next in priority are the critical real-time threads, followed by the noncritical real-time threads, and finally by the non-real-time threads. By default, the garbage collector runs at its initial priority, which is below that of the noncritical real-time threads. But as memory grows short, the VM will boost or raise the priority of the collector to the maximum configured priority.

Figure 4 shows the priority levels for the different types of threads in gray and for the garbage collector at its initial and maximum priorities in pink.

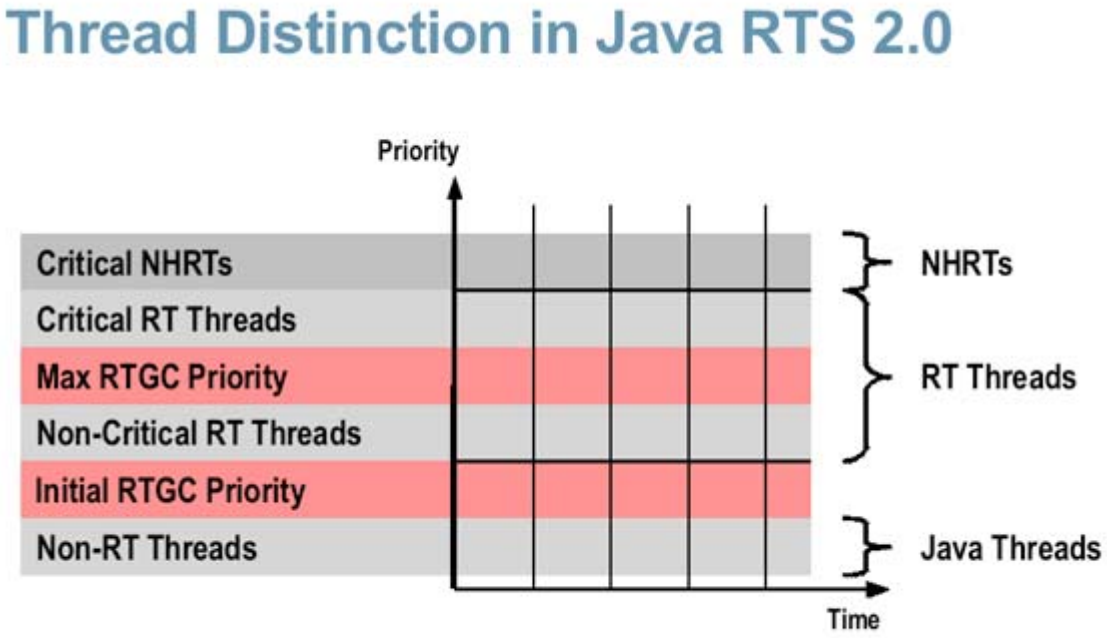


Figure 7.1 thread distinction in java

The important point about the RTGC provided with Java RTS is that it is fully concurrent, and it can thus be preempted at any time. There is no need to run the RTGC at the highest priority, and there is no stop-the-world phase, during which all the application's threads are suspended during GC.

In Figure 5, the RTGC starts executing at its initial priority and is preempted by a high-priority thread. When that thread stops, the RTGC runs again but is again preempted. Finally, if the running threads are allocating memory and the remaining memory becomes low enough, the RTGC is boosted to its maximum priority, where it is preempted only by critical threads.

On a multiprocessor, one CPU can be doing some GC work while an application thread is making progress on another CPU. In Figure 6, the critical NHRTs are running on a separate CPU and therefore do not preempt the RTGC. The RTGC runs to completion without its priority having to be boosted.

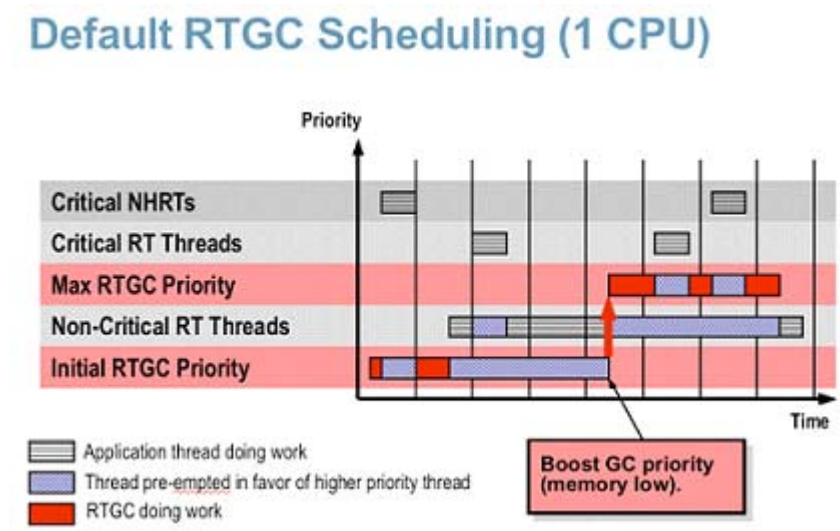


Figure 7.2 default real time garbage collection in one cpu

Thus, the RTGC offered by Java RTS is very flexible. Whereas garbage collectors in other real-time systems usually either have to be executed as incremental, periodic, high-priority activities or else induce an allocation-time overhead, the Java RTS RTGC can execute according to many different scheduling policies.

In addition, instead of trying to ensure determinism for all the threads of the application, the Sun Java RTS team based the default-scheduling policies on Henriksson's algorithm.

The RTGC considers that the criticality of an application's threads is based on the threads' respective priorities and ensures hard-real-time behavior only for real-time threads at the critical level, while trying to offer soft-real-time behavior for real-time threads below the critical level.

This reduces the total overhead of the RTGC and ensures that the determinism is not affected by the addition of new low-priority application threads. This makes the configuration easier because there is no need to study the allocation behavior of an application in its entirety in order to configure the RTGC. Typically, a developer designs determinism by looking only at the critical tasks.

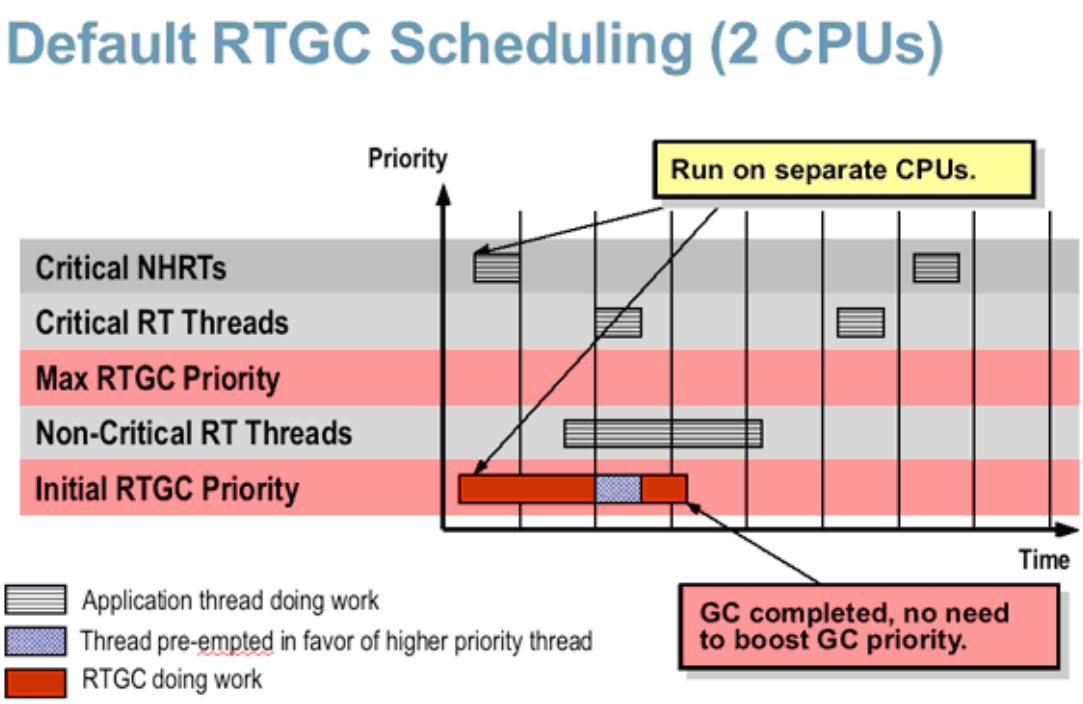


Figure 7.3 default real time garbage collection in two cpus

By setting only two parameters, a memory threshold and a priority, you can ensure that GC pauses will not disrupt threads that are running at critical priority. The big advantage of this approach is that these parameters are independent of the noncritical part of the application. You do not need to reconfigure the RTGC when you add a new noncritical component or when the machine load changes.

The RTGC has an auto-tuning mechanism that tries to find the best balance between determinism and throughput. It tries to recycle memory fast enough at its initial priority for the noncritical real-time threads but without offering any guarantees for them.

If the noncritical load increases, the RTGC may fail to recycle memory fast enough for all the threads and will be boosted to its maximum priority. However, this will not disrupt the critical threads, as long as the memory threshold is set correctly for the application. Only the noncritical real-time threads will be preempted and temporarily suffer from some jitter, or variance in latency, due to memory recycling.

CHAPTER - 8

Implementation and results

Java possesses many advantages for embedded system development, including fast product deployment, portability, security, and a small memory footprint. As Java makes inroads into the market for embedded systems, much effort is being invested in designing real-time garbage collectors. The proposed garbage-collected memory module, a bitmap-based processor with standard DRAM cells is introduced to improve the performance and predictability of dynamic memory management functions that include allocation, reference counting, and garbage collection. As a result, memory allocation can be done in constant time and sweeping can be performed in parallel by multiple modules. Thus, constant time sweeping is also achieved regardless of heap size. This is a major departure from the software counterparts where sweeping time depends largely on the size of the heap. In addition, the proposed design also supports limited-field reference counting, which has the advantage of distributing the processing cost throughout the execution. However, this cost can be quite large and results in higher power consumption due to frequent memory accesses and the complexity of the main processor. By doing reference counting operation in a coprocessor, the processing is done outside of the main processor. Moreover, the hardware cost of the proposed design is very modest (about 8,000 gates). Our study has shown that 3-bit reference counting can eliminate the need to invoke the garbage collector in all tested applications. Moreover, it also reduces the amount of memory usage by 77 percent.

Java application can be made run on the real time operating system. Real time operating system used in this work is sun java mobile embedded system(sun java CLDC)

8.1 Performance comparison of default approach and proposed approach

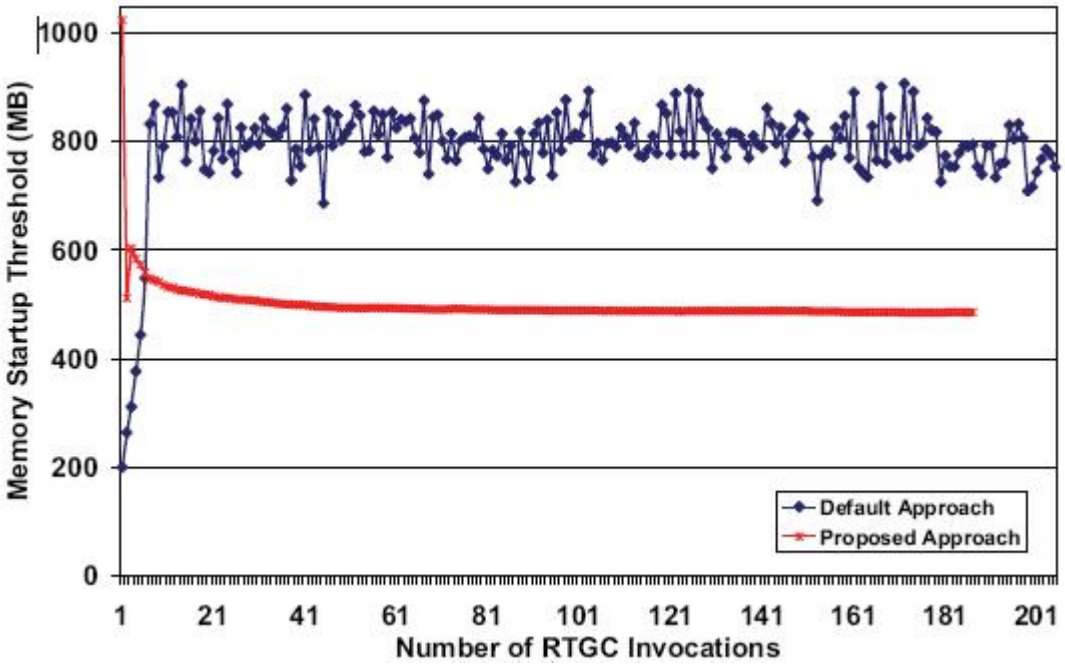


Fig 8.1 performance comparison of default and proposed garbage collection,ref[26]

8.2 Comparison of the garbage collection techniques performances

Garbage collection performances vary when we use reference counting technique. generational garbage collection worst case allocation characteristics are different from reference counting.

	Worst Case Allocation	Worst Case Recycling	generality
1.2.1 Malloc/Free	walk free list	constant	high
Garbage Collection	constant	size of memory (typically)	high
Reference Counting	walk free list	size of memory	med (cycles)
Pool analysis	constant	constant	low

Table 8.1 comparison of garbage collection techniques worst case execution times

8.2 Running a java application on sun java real time operating system mobile embedded application

The java application has been executed on sun java real time mobile embedded application. The application is made run on the simulator of default color phone.the operating system on which the application is working is symbian real time operating system.

The output of the java application on wireless toolkit is shown below

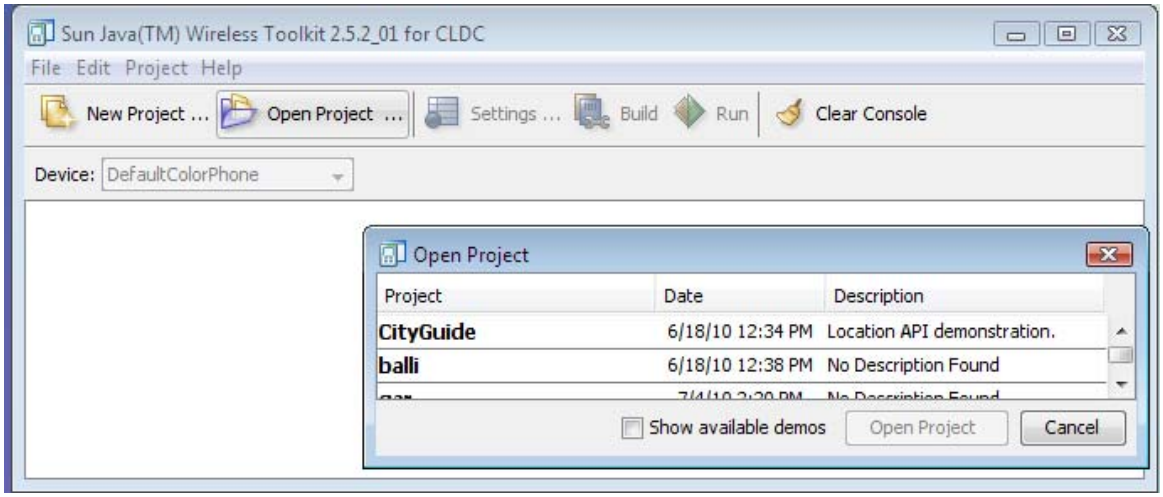


Figure 8.2 opening a project on real time java mobile embedded system

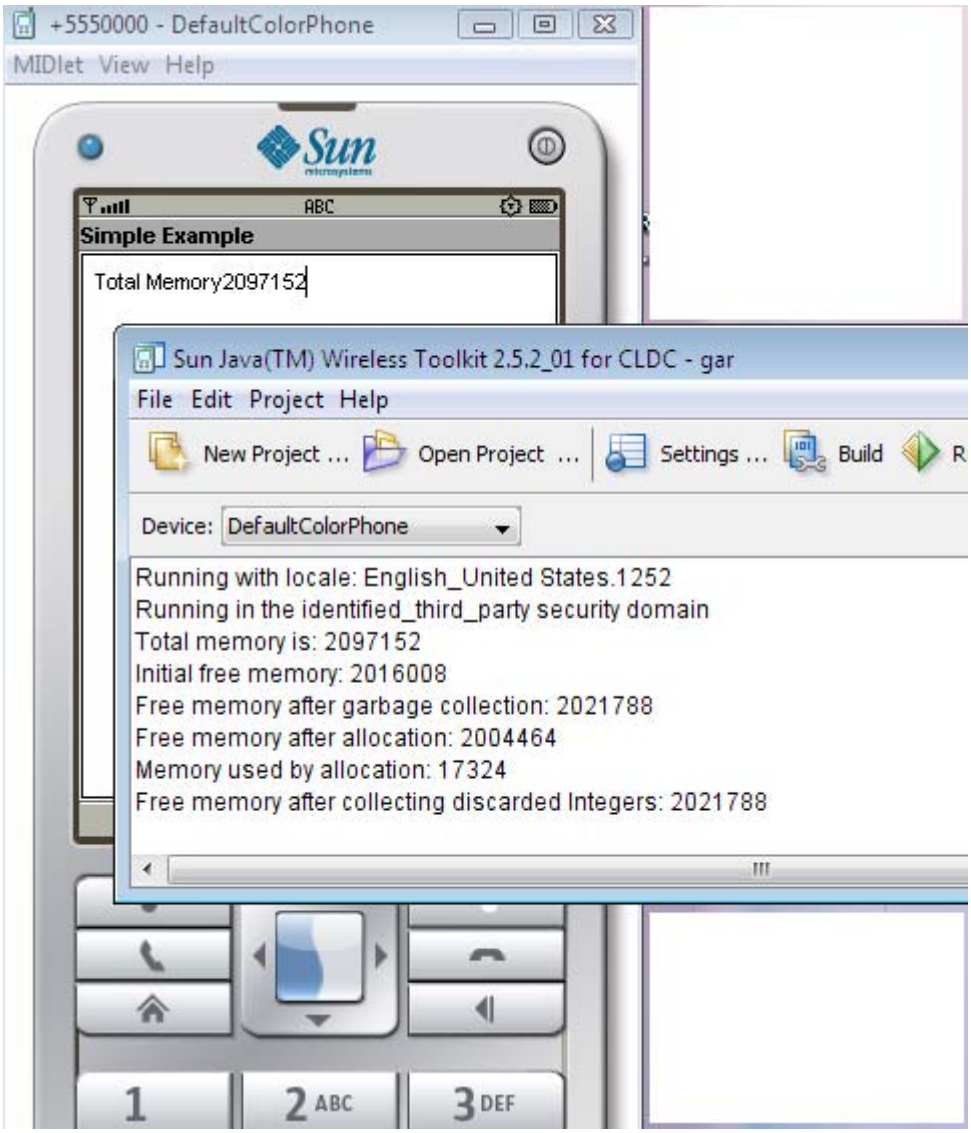


Figure 8.3 executing java application on real time mobile embedded application

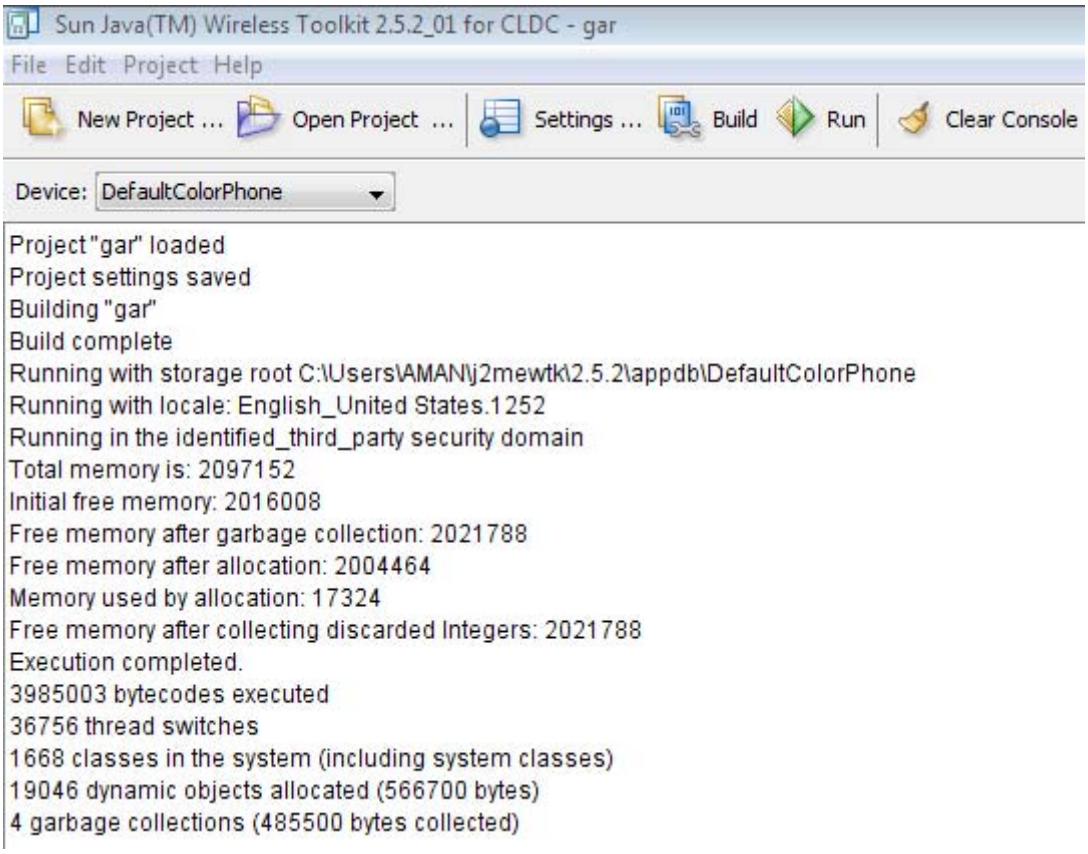


Figure 8.4 result after executing real time garbage collection application

CONCLUSION

As is the case for other programming languages and execution environment, designing and implementing a garbage collector suitable to use in Java embedded systems is not an easy task. It requires a deep analysis of numerous factors ranging from the hardware in use to the dynamic of the application to run on the embedded system. A Java run time capable of instrumenting the memory allocation patterns coupled with the possibility of a pluggable garbage collector seem to be attractive enough, so that several players in the field are seriously considering it as a solution. With more time spent studying the memory-allocation behavior of embedded applications written in Java, we might in the future see the emergence of a collection of ready-to-use garbage-collector algorithms that embedded-system developers will be able to directly incorporate into their design.

The extensions defined by the realtime specification for java enable the use of java in new application domains that have strict requirements on the timing behaviour of the application.

However the need to use special thread classes and specific memory areas that are not under the control of the garbage collector and not accessible by normal threads complicates the development of larger systems that require communication and shared data between real time and non real time code.

The use of the real time garbage collection together with the extensions defined in the real time specifications for java makes it possible to overcome these restrictions and provides a more straightforward and simpler development of realtime code using java. even systems that do not require dynamic memory management within realtime code becomes simpler ,such that higher productivity and higher software quality can be expected. Such a system provides the advantage that made java so successful to the developer of real time systems.

FUTURE SCOPE

As usual in this section an example of things that are of interest will be listed.

Coping garbage collector. Are there programs where a copying garbage collector would outperform a reference counting garbage collector? Do they differ in predictability? These are some of the questions that makes it interesting to make an implementation of a copying garbage collector.. Given that there are some easy switch between the two.

Compiler extension. In order to utilize all strengths of the theoretical aspects he compiler need to be completed with the extensions that are suggested.

Static analysis, Timing analysis, Robustness and Schedulability. These four are bound together. To improve performance static analysis has to be done on the garbage collector. Furthermore, static analysis has to be added to the compiler so that compiled code is analyzed during compile time.

An obvious gain will be that dismissal of unnecessary calls to the incremental and decrement function. Which will improve the efficiency of the reference counter.

Timing analysis is necessary since when otherwise it is not possible to utilize the strength of this system. Without is it is guess which clock speed the processor has to have.

How robust is the garbage collector, is there certain code that makes its functionality questionable? Is it feasible to think that the time during which the system is idle is sufficient for managing the to be free list during asynchronous interrupts?

All these together with a memory analysis will make the base of the static analysis that shows how fast the CPU has to be and how much memory there has to be in the system.

Extensive testing. However, the number of test cases are a bit limited and therefore extensive testing is necessary .If not to see if there are special cases were it can go wrong or at least to analyze behavior during different kinds of code.

Adaptation to different microprocessor. To do this a hardware layer has to be made. This layer consists of an interrupt handler, internal timing and port specific information. It can also contain functions like power saving or other functions that are unique for a specific microprocessor.

REFERENCES

[1] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates ,pages 26-38.ACM Press, 1990.

[2] Paul R. Wilson. Uniprocessor garbage collection techniques. Submitted to ACM Computing surveys 1994.

[3] Yoo C.Chung and Soo-Mook Moon.Memory allocation with lazy fits.

[4] Jaques Cohonu and Alexendru Nikolau.Comaprison of compaction algorithm for garbage collection.ACM transactions on programming languages and systems,5(4):532-553,October 1983.

[5] Ben Cranston and rick Thomas. A simplified recombination scheme for the fibonacci buddy systems, pages 331-332.ACM Press ,june 1975.

[6] David Detlefs.Automatic inference of reference-count invariants.

[7] David M.Harland.REKURSIV:Object-oriented computer arcitecture. Ellis Horwood Ltd.,1998.

[8] Roger Henriksson. Scheduling garbage collection in embedded systems. Phd thesis,Lund Institute of Technology,1998.

[9] Daniel S.Hirschberg. A class of dynamic memory allocation algorithams,pages 615-618. ACM Press , October 1973.

[10] Mark S.johnstone and Paul R. Wilson. The memory fragmentation problem :solved ?October 18,1997.

[11] Donald E.Knuth.The Art of Computer Programming, volume 1:Fundamental Algorithms. Addison-wesley,1973.

[12] Scott Nettles and James O'Toole.Real-Time replication garbage collection. Pages 217-226. ACM Press,1993.

[13] Kelvin Nilsen. High level dynamic memory management for object- oriented real-time systems,pages86-93.ACMpress,1996.

[14] Young Gil Park and Benjamin Goldberg. Static analysis for optimizing reference counting. Information processing letters, 55(4):229-234,august 1995.

[15] Young Gil Park and Benjamin Goldberg. Escape analysis on list. Pages 116-127,june,1992.In proceedings of the 5th ACM SIGPLAN confrence on programming language design and implementation.

[16] James L. Peterson and Theodore A. Norman. Buddy system, Pages 421-431. ACM press, 1997.

[17] Mehran Rezaei and Ron K. Cytron.segregrated binary trees: Decoupling memory manager.

[18] Tobias Ritzau. Memory Efficient Hard Real-Time Garbage Collection. Phd thesis,Linkopings Universitet,2003.

[19] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs.

[20] Wolfram Schutle. Deriving residual reference count garbage collectors July,1994.Sixth international symposium on programming language implementation and logic programming(PLILP),Madrid.

[21] Fridtjof Siebert. Real –Time garbage collection in multi-threaded systems on a single processor.

[22] Fridtjof Siebert. Hard real-time garbage collection. Phd thesis,Universitat Karsruhe,2002.

[23] D. F. Bacon, P. Cheng, and V. T. Rajan. "A Unified Theory of Garbage Collection." In Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems,
Languages, and Applications (OOPSLA 2004), pp. 50 – 68.

[24] H. G. Baker. "The treadmill: real-time garbage collection without motion sickness." ACM SIGPLAN Notices, Volume 27, No. 3, pp. 66–70, 1992.

[25] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko and E. Petrank, "A parallel, incremental, mostly concurrent garbage collector for servers," ACM
Transactions on Programming Languages and Systems (TOPLAS), Vol. 27, Issue 6, pp. 1097–1146, November 2005.

[26] D.F. Bacon, P. Cheng, and V.T. Rajan, “A Real-Time Garbage Collector with Low Overhead and Consistent Utilization,” In Proceedings of the ACM Principles of Programming Languages (POPL 03), pp. 285-298, 2003.