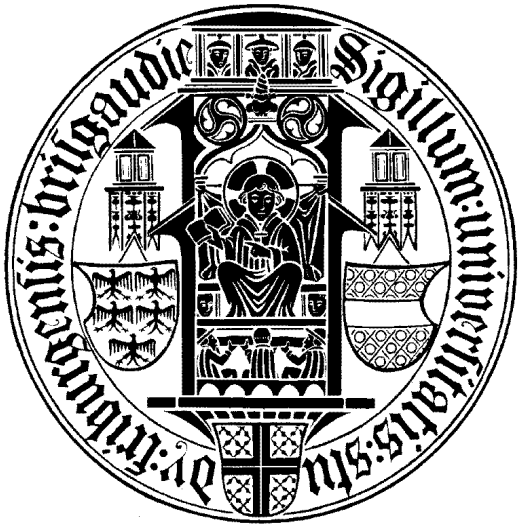
Bachelor's thesis

Garbage collection for sensor motes



Albert-Ludwigs-University of Freiburg
Faculty of Applied Sciences
Department of Computer Science
Chair of Computer Networks and Telematics
Prof. Dr. Christian Schindelhauer

Author: Jan Meyer
matriculation number: 1920463

Supervising tutor:	M. Sc. Faisal Aslam
Examiner:	Prof. Dr. Christian Schindelhauer

submission date: 4. July 2008

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Arbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum Unterschrift

Acknowledgement

I would like to express my appreciation to the people who supported and helped me during my bachelor thesis.

First of all, I would like to thank Prof. Dr. Christian Schindelhauer for his support and refreshing attitude. My supervisor Faisal Aslam was also a valuable source of feedback and constructive criticism and was always available when problems arose.

Next, I would like to thank the members of the TakaTuka project team, without whom this thesis certainly would not have been possible. Especially Gidon Ernst was a driving force and always motivated to work hard for the sake of the whole TakaTuka undertaking.

Not to be forgotten are my friends and student collegues (you know who you are!) for providing some great company during my studies.

Last, but certainly not least, my parents, my sister Nina and my girlfriend Babette have given me endless support over the years. You have my deepest gratitude.

Abstract

The goal of this bachelor's thesis is to provide Java garbage collection and its resulting benefits on sensor motes. In order to program motes, programmers would often have to learn an unfamiliar language. To facilitate mote programming, it is an attractive alternative to program in Java, which is already a widespread and often referred to as an easy to learn language. Furthermore, the Java language comes with integrated garbage collection, which disburdens the programmer from large parts of manual memory management. This thesis examines common garbage collection techniques both in theory and practice.

This thesis emerged from the TakaTuka project, currently in development at the University of Freiburg under the supervision of Prof. Dr. Christian Schindelhauer. Different algorithms were chosen to be implemented and integrated into the TakaTuka virtual machine. Several working garbage collectors are introduced, which are designed to run in environments with very limited hardware resources and keep used memory to a minimum. The proposed garbage collectors are written entirely in Java, but make use of several native methods written in C to manipulate low-level structures in memory and to provide functionality which is not possible to implement in pure Java. The collectors run without occupying a thread of their own.

The most important desired goals to achieve in the described garbage collectors were:

- Garbage collection written entirely in Java
- Low memory requirements
- Garbage collection must be runnable even without thread support

Zusammenfassung

Das Ziel dieser Bachelorarbeit ist es, Java Garbage Collection und die daraus resultierenden Vorteile auf Sensormotes zur Verfügung zu stellen. Um Sensormotes zu programmieren ist es häufig notwendig, dass Programmierer eine ungewohnte Sprache lernen. Um das Programmieren von Sensormotes zu erleichtern, ist es eine ansprechende Alternative dies in Java zu tun, einer Sprache die weitverbreitet ist und oft als leicht zu lernen beschrieben wird. Darüber hinaus beinhaltet Java eine automatische Garbage Collection, die dem Programmierer einen großen Teil des Speichermanagements abnimmt. Diese Bachelorarbeit befasst sich mit verbreiteten Garbage Collection Techniken sowohl theoretisch, als auch praktisch.

Diese Bachelorarbeit ist als Teil des TakaTuka-Projektes enstanden, das zur Zeit an der Universität Freiburg unter der Leitung von Prof. Dr. Christian Schindelhauer in Arbeit ist. Verschiedene Algorithmen wurden ausgewählt, um sie zu implementieren und in die TakaTuka virtuelle Maschine einzubinden. Es werden mehrere funktionierende Garbage Collector-Implementationen vorgestellt, die so gestaltet sind, dass sie in Umgebungen mit sehr beschränkten Hardwareressourcen lauffähig sind und die Auslastung des Arbeitsspeichers auf ein Minimum beschränken. Die vorgestellten Garbage Collectors benötigen keinen eigenen Thread.

Die wichtigsten Ziele, die bei den beschriebenen Garbage Collectors angestrebt wurden, waren:

- Garbage Collection ist vollständig in Java geschrieben
- Geringe Speicheranforderungen
- Garbage Collection muss lauffähig sein, selbst wenn von der Java Virtual Machine keine Threads unterstützt werden

Contents

1 Introduction

1.1 Structure

This thesis is divided into six main chapters. The first chapter lays out the motivation for this thesis. The second chapter gives an overview over the concept of garbage collection in general and the difficulties commonly associated with garbage collection. Furthermore, the chapter aims to deliver insight into the task of implementing the Java language in resource-constrained environments. The third chapter explains a selection of established methods to realise garbage collection and some of the more important variant techniques. The fourth chapter elaborates on the different garbage collectors that were implemented during the course of this thesis. Chapter five provides results gained in the course of test setups of the different implementations, and compares the results in terms of performance. The sixth and final chapter summarises the thesis, discusses weaknesses of the thesis and gives ideas for future work.

1.2 Motivation

With decreasing prices, small hardware devices like sensor nodes - also called *motes* - become increasingly widespread. Java underwent a similar development and is widely accepted as a convenient and uncomplicated language. A downside is that many of the mentioned mote devices are difficult to program and require the programmer to learn a specific language, such as nesC [5], which can be hard at times for programmers to learn. Consequently, it is an attractive alternative to be able to program motes in Java, and many programmers have established Java programing knowledge. It would also be convenient to provide a Java virtual machine written in Java, so that it can be easily extended or ported to other platforms.

The combination of a Java virtual machine on sensor motes appears to be a perfect combination, but there are several difficulties in bringing Java to platforms with restricted hardware. Java is a high-level language with focus on portability and security, but it does not allow the user to directly access the hardware. Every Java virtual machine needs to be able to

access memory directly, as well as the garbage collector as an integral part of the Java virtual machine.

Besides this, the standard Java virtual machine is designed to work on common PCs that have a lot of memory, or Random Access Memory (referred to as RAM), at their disposal. Although there is also the Java Micro Edition (referred to as *J2ME*), which targets mobile devices like modern cellphones, the minimum requirements are still a limiting factor. The *CLDC* (Connected Limited Device Configuration) is one of the two available configurations for J2ME and has a typical memory requirement of approximately 160 KByte of memory [1]. The other configuration, the *CDC* (Connected Device Configuration) has even higher requirements. In comparison to modern PCs, this may seem extremely low, but hardware platforms like the Mica2 mote (see the description of the Mica2 mote in section 1.3) have even bigger resource constraints.

This thesis proposes Java garbage collectors, suitable for Java virtual machines that target small mobile devices with very limited memory resources, such as the Mica2 platform.

1.3 TakaTuka

TakaTuka is the name of a project currently in progress at the University of Freiburg. The goal of the project is to provide a CLDC compliant Java virtual machine suited for small mobile devices with extremely constrained hardware resources. Although the TakaTuka virtual machine is device independent, the device used to develop the TakaTuka Virtual Machine is the Mica2 mote. The platform contains the ATmega128 microcontroller with its main feantures:

- 128 KBytes of Flash program memory
- 4 KBytes of EEPROM
- 4 KBytes of internal SRAM

The TakaTuka virtual machine runs directly on the hardware without an underlying operating system (sometimes referred to as *running on bare metal* [12]). Another important charasteristic of the TakaTuka virtual machine is that it wraps method invocations into *stackframes*,

which are handled by the stack. Each stackframe thus represents one method invocation and has great variation in size, depending on the specifid method being called. The garbage collectors in this bachelor's thesis were integrated into the TakaTuka virtual machine, and TakaTuka was used to test the implementations and to produce the results.

2 Garbage collection

2.1 Introduction to garbage collection

The term garbage collection describes an automated background process, which frees previously allocated memory that cannot be used anymore and is therefore considered "garbage". Languages like C or C++ require the programmer to keep track of the particular memory usage and it is his responsibility to correctly and explicitely free memory which is not required anymore. As many language constructs typically have a very limited lifetime, garbage in the sense as mentioned above accumulates quite quickly. If the programmer neglects the deallocation of unused memory, this will lead to *memory leaks* and is considered a flaw of the program in question. If a memory leak appears, a portion of memory is unintentionally occupied and not available for further use [7, 8]. If repeated memory leaks occur (as for example in every iteration of a loop construct) memory will eventually exhaust [11, 8]. The running program will at the least run with reduced performance, but - depending on the specific program - other symptoms of unwanted behaviour like program crashes are also possible. Obviously a memory leaking, and therefore unreliable program is intolerable in many cases.

In contrast to this, freeing up memory by mistake although it is still in use results in a *dangling pointer* [7, 8]. A dangling pointer is a reference to a memory location which has unpredictable content. If the program uses this random content, this will cause potential unpredictable behaviour of the program itself.

To facilitate programming, the Java language provides integrated garbage collection. Although the existence of a garbage collector in any Java virtual machine is mandatory, there are no restrictions on *how* the garbage collector performs his task. Sun's specification of the Java virtual machine states:

> *"Heap storage for objects is reclaimed by an automatic storage management system (known as a garbage collector); objects are never explicitly deallocated. The Java virtual machine assumes no particular type of automatic storage management system,*

> *and the storage management technique may be chosen according to the implementor's system requirements."* [10]

Although a programmer using an arbitrary Java virtual machine can safely assume that a garbage collector will run in the background, he can not make any further predictions about the speed or working principle of the garbage collector. This means that it is impossible to predict when the garbage collector is active and releases memory. Even a program run without the garbage collector being active is possible. In addition to this, the standard Java virtual machine does not provide real-time garbage collection. Garbage collection partially takes away the responsibility from the programmer of explicitely releasing allocated memory; the programmer does not have to care about explicitely releasing allocated memory. While this is an integral concept of Java and aims at resolving some of the burdens commonly associated with programming, it does not come without disadvantages.

Garbage collection usually imposes computational overhead as well as memory overhead, thus making the application slower and more memory expensive, although it is possible that a sophisticated garbage collection implementation in fact reduces runtime over manual memory management [2].

It is important to know that garbage collection can solve memory leaks and dangling pointers, but there are still some memory-related problems which garbage collection cannot solve. A *logical* memory leak (in contrast to the above mentioned *physical* memory leak) occurs if memory is still referenced, but in fact is never used [11]. Traditional garbage collection cannot detect this and programmers of garbage-collected languages like Java should be well aware of this situation.

3 Garbage collection techniques

Garbage collection traditionally follows one of the three trails described in this section [7, 3]. Sometimes a combination is used. Please note that there are countless variants of garbage collectors and it is impossible to describe them all. This section is a mere introduction of the most important ones. For a more elaborate and comprehensive explanation of the fundamentals and methods described in this section, refer to the book *Garbage collection: algorithms for automatic dynamic memory management* by Jones and Lins [7], which gives detailed insight into garbage collection and the multiple variants.

3.1 Concept of garbage collection

Primitive datatypes like `int` and `boolean` in Java are pushed onto the stack of the corresponding thread. When an *object* is created, memory will be allocated dynamically and the object is stored in the heap. The actual position in memory is defined by the JVM's underlying memory management system [10]. The reference to this object also goes to the stack. The heap itself is shared by all threads.

As mentioned above, a garbage collector runs in the background of the actual program (commonly referred to as the *mutator*, because it alters the memory state) and is supposed to recognise when a portion of heap memory is no longer in use, and subsequently release these memory areas. As the garbage collector has no way of telling if allocated memory *will* in fact be used in the future, it therefore performs its work by searching for parts of heap memory that *cannot* be used again. To satisfy this condition, allocated memory must be unreachable, that is all references to such memory regions have been removed. This can and will frequently occur, for example if a local variable becomes out of scope, or if the last remaining reference to an object is set to another object.

Formally, the set of reachable (or *live*) memory objects, also called *nodes*, is defined as [7]:

$$\text{live} = \{N \in Nodes \mid (\exists r \in rootset.r \rightarrow N) \vee (\exists M \in live.M \rightarrow N)\}$$

In this case, "$\rightarrow$" represents a *"points-to"* relation. (See also section 3.3 for a definition of *rootset*)

The set of unreachable or garbage objects is then simply $live^C$.

3.2 Technique 1 - Reference count garbage collection

One common technique to perform garbage collection is the method of reference counting. As the name suggests, each object always carries the number of references pointing to itself. If this number reaches zero, no references point to the object in question anymore. Thus it is unreachable and can be deleted safely.

3.2.1 Advantages of reference count garbage collection

1. The biggest advantage of this technique is its simple and straightforward principle. In a naive way, it is very easy to understand and implement.

2. With a reference count approach the garbage collector is able to reclaim unused memory immediately after it has become unreachable. Whenever the reference count of one object reaches zero, a small garbage collection can be triggered which reclaims the memory that was occupied by the now unreachable object. Usually, this results in low pause times for the mutator.

3.2.2 Flaws and drawbacks of reference count garbage collection

1. Memory overhead is introduced because every object has to contain the reference count number. As this is necessary for each object, the memory overhead quickly sums up to a relevant magnitude. Although the number of references to one object will generally

tend to stay low, it is still unbounded and can potentially grow towards infinity. One cannot predict how many references will be created pointing to any given object, therefore the total memory overhead is also unbounded. There are several approaches to reduce the memory overhead of reference count garbage collection, like for example the *limited field* approach [6, 13, 7].

2. To keep an accurate count of references, every operation that changes the number of references for any object must be immediately followed by an update of the corresponding reference count number. Frequent update operations are the result and must also be considered to be an important factor in execution speed.

3. If the reference count for one object reaches zero, it can and should be deleted to regain memory. This deletion process can cascade. An example is a chain of objects, with each object being the only reference to the successing object. With the first object's reference count number set to zero it gets deleted and its successor object now has no references anymore, and subsequently also gets deleted. This deleting cascade travels through the whole chain. See figure 1 for an illustration of this situation.

   This means, that even one single changing reference can result in much work for the garbage collector.

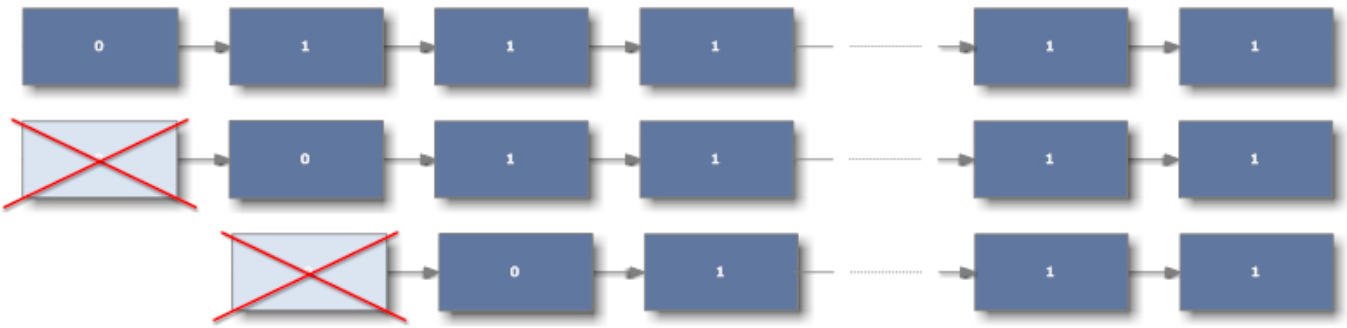


Figure 1: Cascading deletion chain

4. Another important flaw of the reference count technique is that it is unable to detect cyclic reference structures. If two or more objects reference each other in a cyclic manner, the reference count for all involved objects will never reach zero, although they

may be unreachable as a whole. The corresponding memory regions will never be reclaimed and the result is a memory leak. This is one of the reasons that some garbage collectors rely on a reference count collector in combination with another technique to detect all of the unreachable memory regions.

5. Heap fragmentation will occur if there is no additional mechanism to conquer it, as deleted objects will form gaps in the heap.

3.3 Technique 2 - Tracing garbage collection

Another traditional technique is the tracing garbage collection. In its most naive occurence it is implemented with a *stop-the-world* approach, meaning that whenever the garbage collector starts to run, it stops the mutator and reclaims all unused memory at once. This can occur for instance if the whole memory is exhausted and memory has to be freed in order to enable the mutator to continue running. This garbage collection technique is divided into a *mark* phase and a *sweep* phase, therefore a tracing collector is often called a mark-and-sweep collector.

First of all, the term *root set* has to be introduced to the reader. The root set is a specific set of objects, referred to as *roots*, containing all directly reachable objects, such as static objects. It is the entry point for the tracing garbage collector. From there on, it transitively follows all references to all other reachable objects and marks them as being *reachable*, by applying a depth-first search. In the succeeding sweep phase, all unmarked objects are purged from the memory.

3.3.1 Tricolor marking

Tricolor marking is a method to mark the state of an object with respect to the garbage collection [4] and is widely used in tracing garbage collectors. Three different colors are used to mark objects: white, grey and black. Accordingly, the white[1], grey and black set is

[1] also: the condemned set

defined as the set of objects, which possess the specified coloured marking. Please note that an object must belong to one of these sets and that the three sets are disjoint.

- White - the object was not (yet) visited by the garbage collector and is also called *unmarked*. When the marking phase is over, all white objects are regarded as garbage.
- Grey - in order to gain this color, an object must have been *reached* by the garbage collector. It is therefore marked as *reachable*.
- Black - if an object is grey and all its referenced objects are already marked black, the object itself is marked black. Equivalently, a black object is called a *live* object.

There are several ways of realising a tricolor marking scheme. For this thesis, the following scheme was used: initially, the black set is empty. The grey set consists of objects that are directly reachable (i.e. they are either local variables currently in scope, class variables or static objects). The white set contains all other objects.

Two invariants are introduced and are conserved during the garbage collection process:

1. During one marking phase, an object never changes its marking to a lighter color. White is defined as being lighter than grey, grey is defined as being lighter than black.
2. A black object never references another object marked with a lighter color.

When the grey set is empty, the marking is over and each object is either marked as *live* or as *garbage* and can be treated accordingly by the garbage collector. See figure 2 for an example of the tricolor marking scheme.

3.3.2 Advantages of tracing garbage collection

1. The biggest benefit gained from using a tracing garbage collector over a reference count collector is, that it is not affected by cyclic structures. The collector can free all unreachable memory regions without special precautions.
2. In comparison to reference count garbage collection the memory overhead is lower, as the marking of an object in its simplest form only takes up one single bit per object.

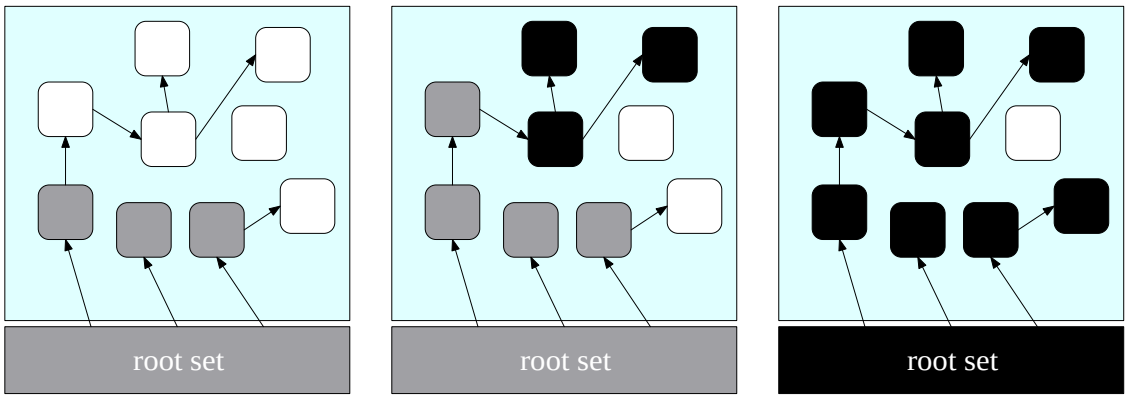


Figure 2: Example of a tracing garbage collection run with the chosen tricolor marking scheme

3.3.3 Flaws and drawbacks of tracing garbage collection

1. When implemented as a stop-the-world collector, a considerable pause in the mutator is the result. The point in time when the garbage collector comes into effect is usually unpredictable and its running time unbounded. This is obviously intolerable in real-time applications and inconvenient in other scenarios at the least.

2. Serious running time overhead is introduced, because a (naive) tracing garbage collector has to identify all reachable objects before it can free unreachable memory regions.

3. More problems are introduced, when this technique is not implemented with a stop-the-world approach and the mutator and the collector interleave. If the mutator adds new references, these can go by unnoticed. The newly referenced objects could stay unmarked and would be falsely purged from memory, depending on the point in time when this situation occurs. It is also possible that references are removed, but the now unrechable objects will not be reclaimed.

4. With the tracing approach, heap fragmentation is also a problem. If the unreachable objects are simply erased from the memory, this will create gaps.

3.4 Technique 3 - Copying garbage collection

If heap fragmentation is an issue, a garbage collector following the copying approach might be the best solution. Usually, in this technique the heap is implemented as two equal-sized *semi-spaces*. Objects are only allocated in one of the two partitions, commonly called the *from-space*, while the other one is kept free. If a garbage collection is triggered, the surviving objects are copied to the other heap partition, the *to-space*, so that they form one contiguous block. Subsequently, the from-space is entirely erased and the roles of from-space and to-space switch. The actual collection process is implementation dependent, but often realised as a tracing algorithm.

3.4.1 Advantages of copying garbage collection

1. New allocations can be managed easily, as the used memory region is always one contiguous block. New objects can be conveniently allocated at the end of this block, requiring just one pointer marking the end of the used memory region.
2. Heap fragmentation does not occur, because used memory is always kept contiguous.

3.4.2 Flaws and drawbacks of copying garbage collection

1. Serious memory overhead is introduced, as one semi-space of the heap is always kept free.
2. Additional effort is necessary to copy the surviving objects to the to-space, thus increasing the runtime of the garbage collector.
3. The copying approach only addresses heap fragmentation, but an actual collection algorithm is needed to determine the reachable objects.

3.5 Important variant techniques

In this section, a selection of the more common variant techniques are briefly described. Usually, they are integrated as part of one of the basic techniques which are described above

and enhance their functionality.

3.5.1 Compacting variant

In order to handle heap fragmentation in garbage collection algorithms a common technique is to compact the heap immediately after the collection process. This can be achieved by sliding the live objects to the lower end of the heap to form one contiguos block. Obviously, this additionaly degrades performance.

3.5.2 Incremental garbage collection variant

In order to divide the garbage collection into smaller portions, this variant is used. As a result, the pause times for the mutator are kept low, because most of the time only a partial garbage collection (called a garbage collection *increment*) is performed, rather than a full garbage collection (called a garbage collection *cycle*). However, sophisticated mechanisms are needed to ensure that the memory is in a consistent state after one increment is processed. One common way to achieve this is the introduction of *write* or *read barriers*.

3.5.3 Generational garbage collection variant

To avoid an examination of all objects, the usually low lifetime of objects is exploited in this variant. Again, the heap is divided into several not necessarily equally-sized partitions. Objects are created only in one partition, commonly referred to as *eden*. This represents the youngest generation of objects. After a garbage collection on this partition is complete, the surviving objects are then copied to another partition, representing an older generation. As investigations in this area show [9], most of the garbage objects are relatively young objects. By taking advantage of this, the rest of the heap where older generations are located, does not need to be visited until memory is completely exhausted. Only then, a complete garbage collection over the entire heap is necessary. In a way similar to the techniques mentioned above, the advantages of this variant are traded off by imposing memory overhead and additional computational effort.

3.6 Review of the garbage collection techniques

As the preceeding section shows, there is no technique which suits every situation or environment ideally. According to the requirements and emphases a suitable garbage collection algorithm has to be chosen. Every traditional garbage collection method has its drawbacks that have to be considered carefully.

To sum up the three traditional techniques, it can be said that a reference count algorithm cannot detect all types of garbage. It is therefore a bad choice for restricted environments, like sensor motes, because a fallback algorithm has to be implemented to detect cyclic structures, or one has to hazard the consequences of not being able to free all garbage memory. Memory has to be conserved at a very high priority, so usually additional mechanisms (which on the other hand increase the size of the garbage collection code iteself) have to be implemented.

When comparing a tracing collector with a copying collector, Jones and Lins [7] make an important observation. The performance of the two collector types is based on the the amount of reachable memory divided by the size of the heap. They define:

efficiency $e =$ The amount of memory reclaimed in a unit time

Residency $r = \frac{R}{M}$, with R being the amount of reachable memory and M being the size of the heap

The two algorithm efficiencies are

$$e_{Copy} = \frac{1}{2ar} - \frac{1}{a} \qquad e_{Mark-and-sweep} = \frac{1-r}{br+c}$$

, where a, b and c are constants with the property $a > b > c$.

As a conclusion from this, it can be said, that if on the one hand the heap can be made large enough, to keep the residency low, a copying collector is more efficient. On the other hand, if the residency exceeds a certain threshold r^*, then a mark-and-sweep collector is more efficient[7].

Especially in situations where memory is scarce, such as on sensor motes, the heap cannot

be made very large due to the hardware limitations, so the residency will generelly be fairly high. Hence, a mark-and-sweep collector may perform better and will serve as the basic technique for this thesis.

3.7 Real-time garbage collection

Real-time garbage collection is not a technique of its own, it is a matter of implementing a garbage collector algorithm in a way that it supports real-time performance. Real-time, often confused with quickness, does in a way mean the opposite. Overall speed is reduced to enable the mutator to meet its timing deadlines. This may not seem perspicuous at first glance, but one has to distinguish between the complete runtime of a program and isolated pieces of runtime. The basic idea is that in every logical unit of runtime, the mutator is able to achieve a certain defined amount of work. This is only possible if the garbage collector does not delay the mutator and forces it to miss its deadline. In order to achieve this, a real-time garbage collector has to distribute the whole garbage collection work into small parts, which can be quickly processed. Real-time performance further is split into two classes: In *soft* real-time, the missing of deadlines is undesired, while in *hard* real-time it is considered utterly unacceptable.

Consider, for instance, a program that captures picture frames sourced by a digital movie camera and transfers them to a PC. For every frame, which represents a logical unit of the complete task, the mutator must be able to fully process it, otherwise frames would be missing. This is but one example where real-time performance is at least useful, if not necessary. It is left to the reader to imagine scenarios where the missing of such deadlines could for example endanger humans or damage expensive machinery.

4 Garbage collection implementations

During the course of this bachelor thesis four different garbage collectors were implemented. In this section the alogrithms and implementations are described in detail. The reader is advised to keep in mind, that a custom-built Java Virtual Machine was used as a basis for this work and some things may behave differently as accustomed. See section 1.3 for more details about the TakaTuka virtual machine.

4.1 Shared concepts between all implementations

In this section, some fundamentals which are true for all implementations in this thesis are described.

4.1.1 Memory allocation

Sensor motes in general have very low memory available. To prevent polluting the already scarce memory, memory allocation is kept simple and a sophisticated memory management scheme is abandoned.

The available RAM contains both the heap and the stack, with - in the case of this thesis - the heap growing from low to high memory space and the stack from high to low memory space towards each other. New objects are always allocated at the high end of heap memory. For this purpose the TakaTuka virtual machine maintains a pointer specifying this position.

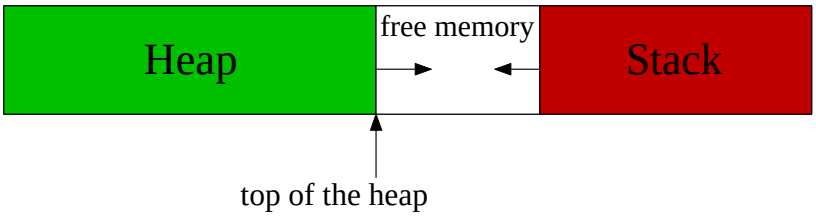


Figure 3: Memory management layout used for this thesis. A pointer to the high end of heap memory is used as the position for the next allocation.

4.1.2 Integration of garbage collection into the virtual machine

All garbage collectors are integrated into the virtual machine in a way, so that the current garbage collector, which can be selected at compile time, is automatically triggered when free heap memory is insufficient to complete the next object allocation. Additionaly it is possible to manually call the garbage collector, by invoking `System.gc()`. Unlike in standard Java this method imediately initiates a garbage collection. Also, it is important to note, that there are currently no other functions for automatically triggering the garbage collection than the one described before.

Since the garbage collection is entirely written in Java, some interface to C must be provided to realize low-level functionality like direct memory access. This is done with the use of native methods in Java calling the corresponding C-method.

4.1.3 Usage of the root set object

All implementations use a root set, which includes all static class variables (which are reachable at all times, once the class is loaded), as well as all variables that are in scope at the time, when garbage collection starts. These objects are gathered by examining the structure of called methods. From the current method back to the downmost calling method, each stackframe representing these methods are traversed to determine the root set members. The root set object is dynamic so that it always contains the correct members at all times.

The root set object is used to iterate over all objects it contains. These objects are, following the definition of the root set (see section 3.3) directly reachable. By following the references from these objects transitively, all reachable objects can be determined and are marked for the garbage collector, using the `gc_info` byte. Upon creation of an object it is set to the value of 0, meaning unmarked or unprocessed.

4.1.4 The `gc_info` byte

Each object in the TakaTuka virtual machine carries one byte of information relevant only for the garbage collector. Dependent on the different implementations, it serves several pur-

poses, which are mentioned in the description of the individual implementations. In general, the `gc_info` byte contains information about the state of the object regarding the current garbage collection, as for example the color marking or the affiliation to a generation of objects.[2]

4.2 Implementation 1 - Mark, sweep and compact algorithm

The first of the four implemented garbage collectors follows the tracing approach (See also section 3.3 for more details). This algorithm also conquers heap fragmentation by compaction of the heap. The algorithm itself consist of two phases: the marking phase and the sweep-and-compact phase, implemented as two different Java methods. It makes use of the tricolor marking and the invariants introduced, as described in section 3.3.1

Pseudocode is available in appendix A.1.

4.2.1 The marking phase

In the marking phase all reachable objects must be marked using the `gc_info` byte to hold the three colors as introduced in 3.3.1. This is achieved by a recursive tracing method iterating over all root set members and calling itself for each referenced object. Mathematically it constructs a transitive closure over all reachable objects. When an object is first encountered, its corresponding `gc_info` byte is set to grey, meaning that it should not be traced again in the future. This prevents infinite loops of objects referencing each other in a cyclic manner. If an object marked as grey is encountered, the individual method invocation returns without effect.

If an object references no other objects or if all references have been already processed, the recursion returns and the object is marked black, meaning that the object is completely processed. When the original calling method returns from recursion, all `gc_info` bytes of all objects in the heap are either white or black. The unmarked objects can then be deleted safely, as they are proven to be not reachable. This is done in the next phase.

[2]For the sake of brevity, the *marking* of an object refers to the value of the `gc_info` byte, if not stated otherwise.

4.2.2 The sweep-and-compact phase

In this phase, unmarked objects are deleted from the heap and eventually the heap is compacted by sliding the survivor objects together at one end of the heap. The method iterates over all objects in the heap and - depending on the value of the `gc_info` byte - it takes one of two actions.

If the object is unmarked, it is deleted, thus creating a gap in the heap memory. The method remembers the location of the deleted object. However, if the object is marked, it is moved to a lower memory location, if such an unoccupied location exists. Subsequently, it is set back to white for future invocations of the garbage collector. When this method returns, all garbage objects in the heap are deleted and the heap objects form once again one single contiguos block in memory. It also retains the invariant, that objects in the heap are sorted by their time of creation.

4.2.3 Complexity

The complexity of the mark phase is proportional to the amount of reachable objects in the heap, because the garbage collector only touches objects that are actually present and reachable. If l is the number of live objects in the heap, the complexity of the marking phase is $O(l)$.

The sweep-and-compact phase has a complexity proportional to the number of objects in the heap, because the collector must traverse the whole heap to purge unreachable memory. If n is the number of objects in the heap, the complexity of the sweep-and-compact phase is $O(n)$.

4.3 Implementation 2 - In-place mark, sweep and compact algorithm

This implementation is similar to the first one. It is also a mark, sweep and compact algorithm and can be considered a variant. Since the original implementaion depends on recursion, which lead to many stack operations of pushing and popping stackframes, this implementation solves the marking phase without recursion. This results in a mathematically

worse complexity, but reduces the time-consuming recursive Java method calls and spares the additional stack memory otherwise needed for recursion.

The sweep-and-compact phase is exactly the same as described in Implementation 1.

Pseudocode is available in appendix A.2.

4.3.1 The marking phase

This method uses two pointers `leftEnd` and `rightEnd`, marking the interval of the heap, that has to be examined. First, it marks all objects in the root set as *encountered*. The two pointers `leftEnd` and `rightEnd` are adjusted to contain all marked objects. Eventually, the range between `leftEnd` and `rightEnd` is traversed and each reference from a previously marked object is followed. Again, the new-found objects are marked as *encountered* and the pointers of `leftEnd` and `rightEnd` are adjusted, if an object outside of this range is found. When no new objects are marked the method returns, otherwise the method examines the region between `leftEnd` and `rightEnd` as new objects are to be processed. Throughout the whole method, if all references from one object have been examined, that object is marked as *processed* and is no longer regarded. The sweep-and-compact phase is very similar to Implementation 1. Since markings with slightly different meanings are applied to the objects, the invariants described in 3.3.1 no longer hold. In the sweep-and-compact phase, objects that are marked as *processed* are live objects and will not be erased.

4.3.2 Complexity

The marking phase's complexity is $O(n^2)$, if n is the number of objects in the heap, as the `leftEnd` and `rightEnd` pointers can be changed $0(n)$ times and $0(n)$ objects have to be traced in each iteration.

The sweep-and-compact phase's complexity is identical to Implementation 1. See section 4.2.3 for details about the complexity.

4.4 Implementation 3 - Generational mark, sweep and compact algorithm

As described in the preceding section, a generational algorithm divides the heap into several partitions. For this thesis only two partitions, one for the young generation, one for the old generation are used. The old generation preceeds the young generation in memory space. With this solution, the allocation of objects in the heap does not need to be changed, as allocation will always take place at the high end of heap memory and thus in the young generation space.

If the garbage collection is triggered for the first time, the situation is as follows: Since there are no objects in the old generation, the young generatin must occupy the entire heap. It is then garbage collected using the aforementioned mark, sweep, and compact approach. After the garbage collection is complete, the surviving objects are marked as part of the old generation and are compacted towards the lower end of the heap. Immediately after a garbage collection, there are no objects in the young generation.

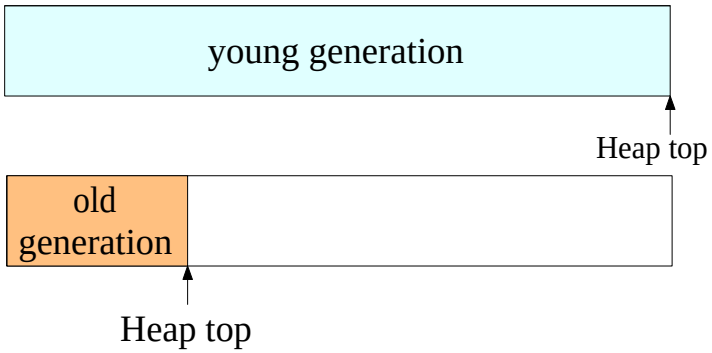


Figure 4: State of the heap before and after the first garbage collection.

With each future garbage collection invokation, it first tries to free memory by applying the mark, sweep and compact algorithm only to the young generation. This is called a minor collection. Since all objects which survived a previous collection are marked as *old*, such objects and references are igored. This also applies to encountered old objects, found via a reference from a *young* object to an *old* one and is not an issue. The other direction is a problem, though. Since it is possible, that *old* objects also reference *young* objects, the referenced *young*

objects must be considered in the garbage collection. Otherwise, they would be mistakenly deleted, although they are not garbage. To conquer this problem a so-called *remembered set* of references from the old to the young generation is introduced. A write-barrier[3] ensures, that all created references which have this property, are put in the remembered set. The remembered set must then be traversed by the garbage collector to find all transitively referenced objects in the young generation. Refer to figure 5 for an example of references across generations.

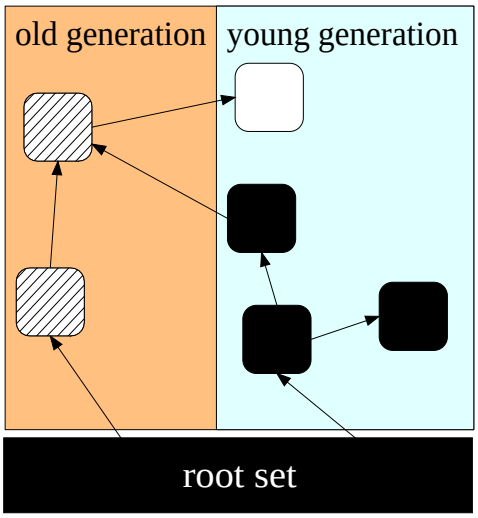


Figure 5: If only the young generation is collected, dashed objects would not be traced without introducing a write barrier. In this case, the white object would then be mistakenly deleted, because it is declared garbage. The introduced write-barrier and the remembered set prevents this situation.

If the garbage collector cannot free (enough) memory by collecting only the young generation, the old generation is added to the collection process. This is equivalent to a garbage collection cycle as described in 4.2, because simply the whole heap is collected and is called a major collection in this case.

Pseudocode is available in appendix A.3.

[3]The write-barrier catches all references at creation time and if they fulfill the mentioned property, they are put into the remembered set

4.4.1 Complexity

In the case of a minor collection only the partition of the heap containining the young generation is collected. The complexity of the marking phase is therefore $O(l)$, where l is the number of live objects in the young generation. The complexity of the sweep-and-compact phase is $O(n)$, where n is the number of objects in the partition holding the young generation.

The major collection is identical to Implementation 1. See section 4.2.3 for details about the complexity.

4.5 Implementation 4 - Reference count algorithm

This implementation follows the reference count approach, as introduced in the preceeding section. To support this, each object now carries its reference count, which is updated when a change to the reference count of an object occurs. If the reference count of an object reaches zero, the object is immediately deleted. Since no further action is taken, this results in a gap in heap memory. The pointer marking the high end of the heap is not adjusted, even if the deleted object was the highest in memory. As object allocation continues to take place at the high end of heap memory, the heap can be recognized as “full”, even though a very low fraction of memory might actually be occupied. To address this issue, the garbage collector simply tries to compact the heap. This can be achieved by a simple linear traversal of heap memory, since all present object in memory are survivors and no markings or other properties have to be examined. This is called a minor collection. See figure 6 for an example of a minor collection.

Pseudocode for a minor collection is available in appendix A.4.

If the simple compaction of the heap fails to provide (enough) free memory, a garbage collection of the whole heap as described in 4.2 is triggered and is called a major collection in this case.

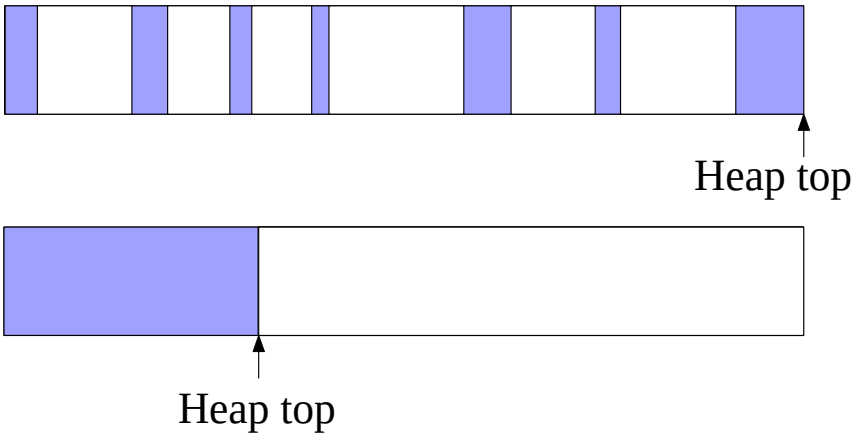


Figure 6: State of the heap before and after the reference count collector compacts the heap.

4.5.1 Complexity

In the case of a minor collection only a compaction of the heap is performed. This requires one single linear traversal of the heap. If n is the number of objects in the heap, the complexity is therefore $O(n)$.

The major collection is identical to Implementation 1. See section 4.2.3 for details about the complexity.

5 Tests

In this section, the tests that were performed to further examine the introduced garbage collection in practice are described. Not presented here are the various test setups, where the correctness of the different implementations was examined and confirmed.

To make the results as meaningful as possible, different test scenarios were prepared to measure both speed and space characteristics of the four algorithms. In this section, a selection of the performed tests that produced the most meaningful results is presented.

5.1 Fundamentals of the test procedure

In order to perform the tests, different test scenarios were programmed in Java. The TakaTuka virtual machine was used to execute the program and also the introduced garbage collectors. All tests were performed on a Dell OptiPlex PC, with a 3 GHz Pentium 4 CPU and 1 GByte of RAM, running the Debian GNU Linux operating system.

First, a random state of an occupied and uncollected heap is generated. This means, that a number of objects are created in the heap. With a certain likelihood, an object gets root set membership. Next, the program iterates over the objects in the heap to determine if an object gets references to other objects, again based on a given likelihood. The objects which get root set membership or references are randomly chosen amongst all objects. The referenced objects are also randomly chosen, which can result in cyclic references as well. See figure 7 for an example of cyclic references in the heap. To achieve this, the test scenarios rely on the standard C random generator to produce the needed element of chance.

These test setups, although not necessarily realistic, reliably produce states of the heap where garbage and live objects are at random positions in the heap. These heap states also include important characteristics, such as cyclic references, references of an object to itself and multiple references to the same object.

After the whole object creation process, a garbage collection is forced to detect the garbage objects and clean up the heap. As the objects are created first and get references afterwards, the reference count collector would normally erase all the created objects, as they become

immediate garbage upon creation. In order to keep the results comparable, a mechanism was introduced to prevent the reference count collector from doing so. When the objects are created, a single linked list of references between all objects is constructed to keep them alive. After the final references are set, the linked list is abandoned, thus leaving some objects as garbage and some alive. This was important to retain the same conditions for all the collector implementations.

The parameters for the test setups were:

- Maximum heap size
- Maximum stack size
- Number of created objects
- Approximate percentage of objects with root set membership
- Approximate percentage of objects with references to other objects
- Number of consecutive object creation and garbage collection cycles
- Number of total program runs

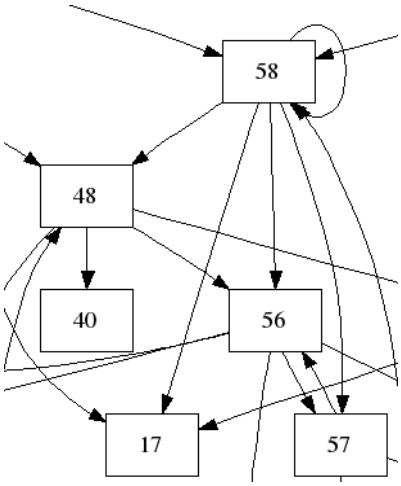


Figure 7: Extract of an automatically generated sample heap state graph showing several cyclic references.

First test runs only took snapshot measurements at certain defined time intervals, for example at every 10 ms. However, as the resulting data clearly showed, the measurements were

too unprecise to draw reliable conclusions. One major issue was, that during a program run several loop constructs are processed, for example numerous invocations of object constructors. If the program was currently inside of a function or that function had just returned when the measurement occurred, had a significant impact especially in the measurements of stack usage. Consequently, this test procedure was abandonded as the yielded data must be considered corrupt.

The whole testing procedure was then changed to log every change in memory usage. This protocol prove to be much more precise and was therefore used to generate the final results that are presented in this section. The downside of this procedure was that vast amounts of data are produced during the tests. Every 10 ms an approximate 4000 data records are generated. This renders the measured data impossible to be processed manually or to be presented in tabular form. Instead, python scripts processed the produced data to bring it into a format for convenient further automatic processing. To reduce the variance introduced by the random number generator, several program runs under the exact same circumstances were measured and the average values were calculated. All tests were performed with 100 program runs to provide the raw data, which was then taken to calculate the average values. The data was then condensed into graphs generated by gnuplot[4] to show the overall development of memory usage in both stack and heap over time[5]. For all the graphs in this section the following abbreviations were chosen:

- MSC - mark, sweep and compact collector as described in section 4.2.
- IP - in-place mark, sweep and compact collector as described in section 4.3.
- GEN - generational mark, sweep and compact collector as described in section 4.4.
- REF - reference count collector as described in section 4.5.

[4] gnuplot is a command-line driven plotting tool, `http://www.gnuplot.info`

[5] As the graphs show the average values, some curves might look different as expected. This is to be ascribed to the different values of individual program runs at the same point in time.

5.2 Tests with 4 KByte of total memory

To simulate a resource-constrained hardware device like the Mica2 mote, several tests were performed with a total of 4 KByte of total memory. As the Mica2 also possesses 4 KByte of RAM, it is likely that the presented results are similar to test results actually produced on the Mica2 mote. Tests directly on the Mica2 mote were not possible due to the unfinished status of the used TakaTuka virtual machine.

The total of 4 KByte memory must be distributed to the heap and stack size. Experiments showed that a maximum heap size of three quarters and a maximum stack size of one quarter of the available memory prove to be a stable setting. In absolute numbers, the maximum heap size was set to 3 KByte and the maximum stack size to 1 KByte.

5.2.1 Test scenario 1 - high heap occupation with low reference density and low memory availability

The first performed test series under the circumstances described above, had the following parameters:

- 120 objects in the heap
- 5 % of all objects have root set membership, approximately
- 10 % of all objects have references to other objects, approximately
- 1 cycle of object creation with succeeding forced garbage collection
- 100 program runs

Figure 8 shows the total memory usage in this test. The different phases of the test run - object creation, garbage collection computation and freeing of memory can be clearly distinguished as the steep rising, the plateau and the steep falling of the curves, respectively. What can be observed is that both the generational and the reference count collector occupy significantly more memory, as is described in 4.4 and 4.5.

Also, the longer runtime of the in-place collector, due to the quadratic complexity is clearly evident. This collector also shows a very long transition from the plateau to the falling curve at the end. As the runtime of the algorithm is dependent on the order of references, few program runs finish very quickly, while most of them take a significantly longer time to end. When calculating the average values for all runs, this results in the mentioned long transition.

Although the reference count collector starts to free memory early, it takes longer to finish this phase, due to the recursive resolving of cascading references, observable in the shallower falling curve at the end.

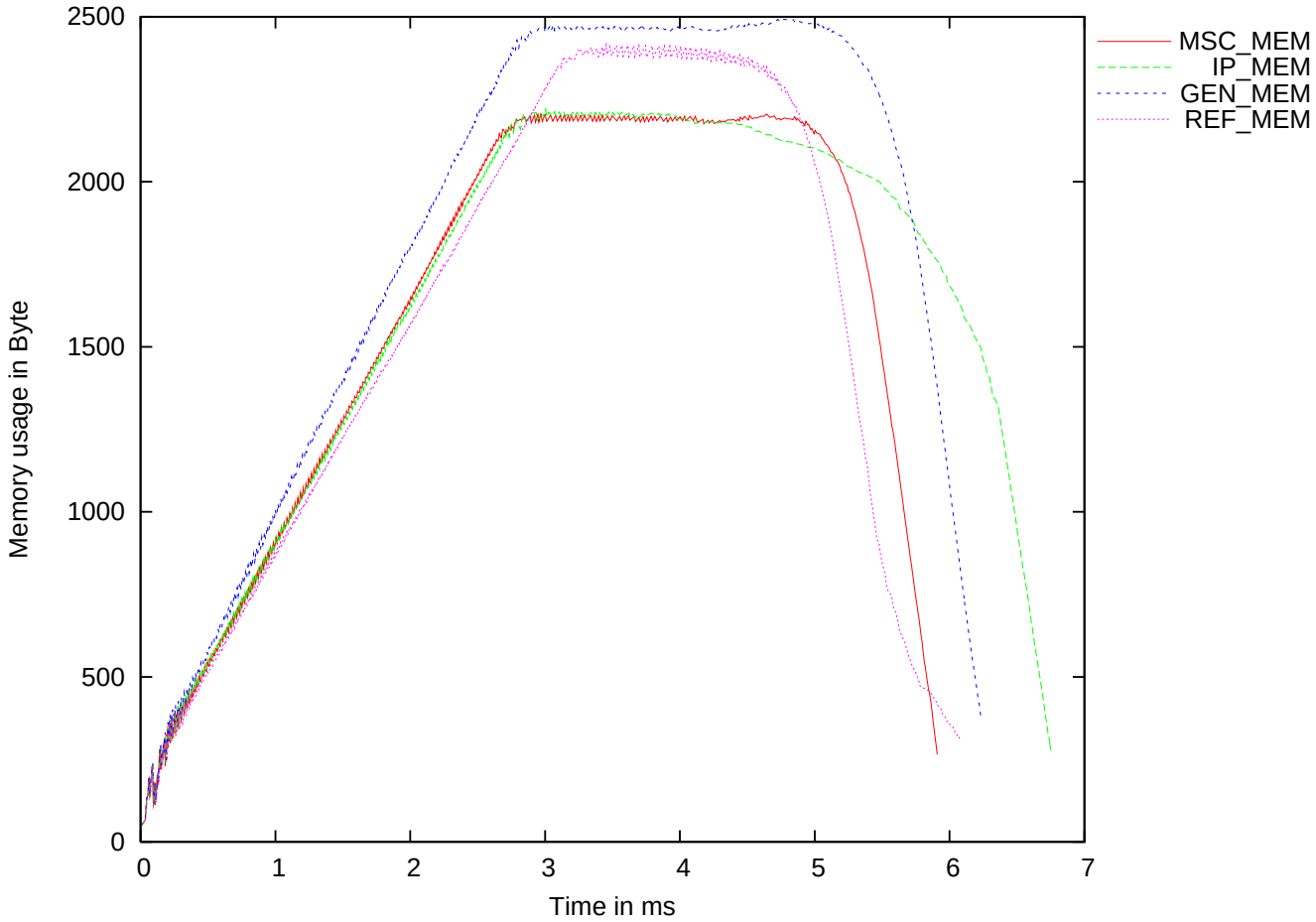


Figure 8: The total memory usage in test 1

5.3 Tests with the maximum amount of memory

To further expose the differences between the different collector implementations, tests were performed with larger memory for the heap and stack. Due to limitations in the TakaTuka virtual machine 65535 Byte ($2^{16} - 1$ Byte) is the maximum amount of memory that can be addressed. To examine the behaviour of the introduced algorithms with more memory available, several tests were performed with 8192 Byte assigned to the maximum stack size and the remaining 57343 Byte of RAM assigned to the maximum heap size.

5.3.1 Test scenario 2 - high heap occupation with high reference density and maximum memory availability

The following parameters were chosen for this test:

- 3000 objects in the heap
- 5 % of all objects have root set membership, approximately
- 40 % of all objects have references to other objects, approximately
- 1 cycle of object creation with succeeding forced garbage collection
- 100 program runs

Figure 9 shows the total memory usage in this test.

As expected, the characteristics from the previous tests remain, but one striking difference is shown. Both the mark, sweep and compact and the generational mark, sweep and compact algorithm possess a distinct bump at the end of the curve plateau. Further inspection delivered that this is caused by the increased amount of references in the heap. Because of the reference density, the average maximum recursion depth, that the collectors had to handle rose to a value of slightly over 60. The two implementations relying on recursion had to build up many stackframes, representing the recursive method calls before being able to resolve the recursion. Thus, the bump is caused by an increased usage of stack memory, as is also shown in figure 10.

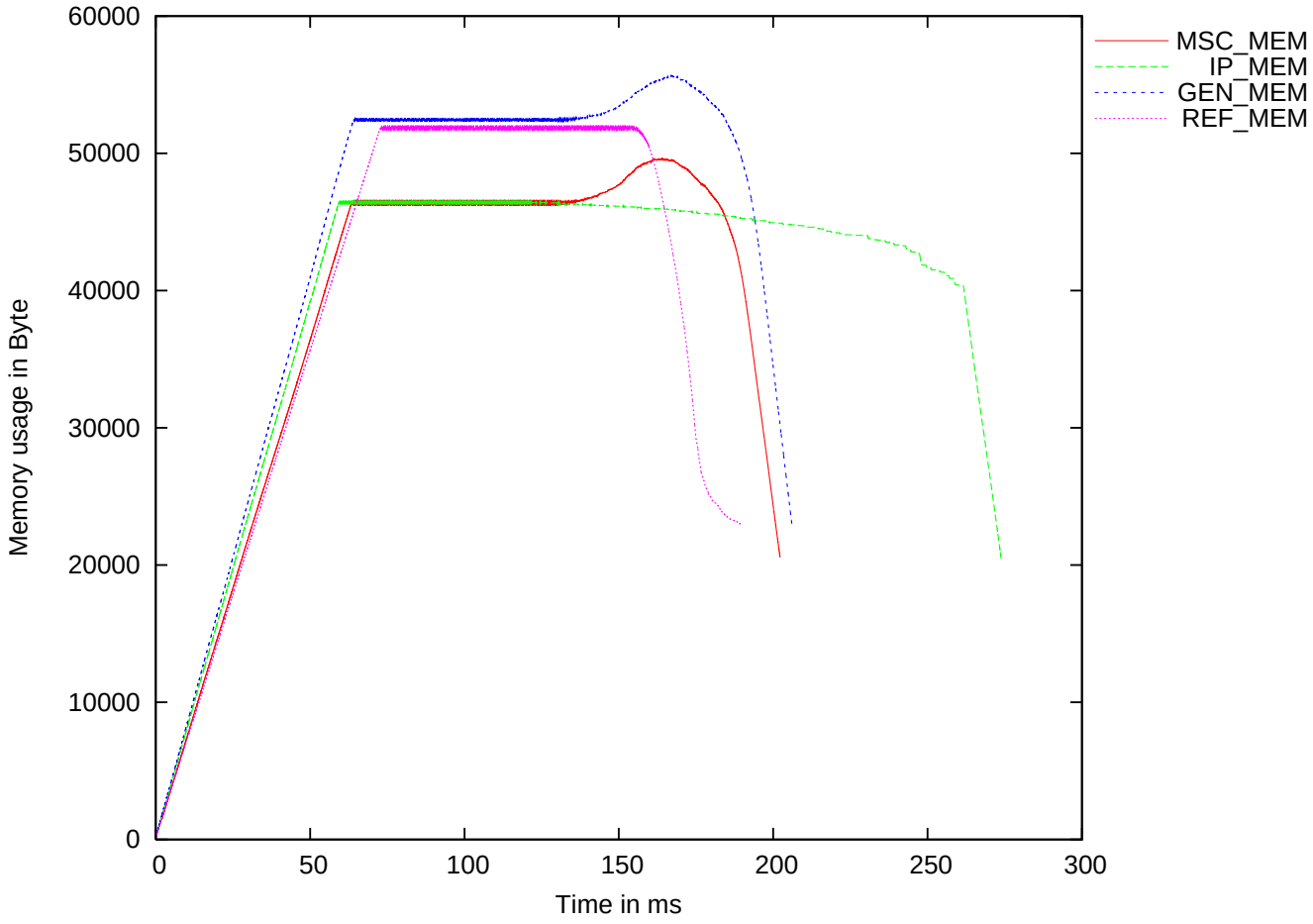


Figure 9: The total memory usage in test 2

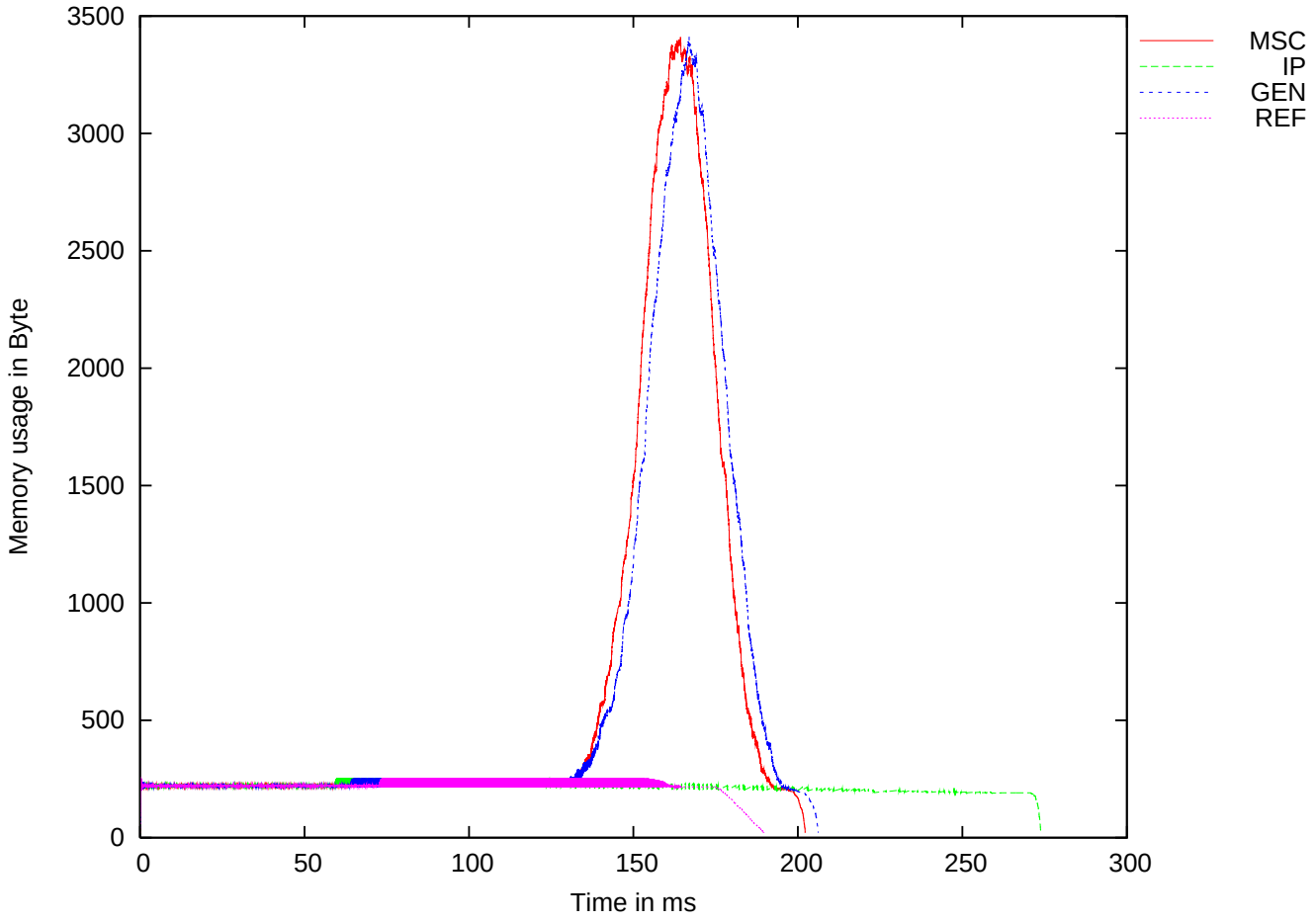


Figure 10: The stack memory usage in test 2

5.3.2 Test scenario 3 - low heap occupation with low reference density, maximum memory availability and several consecutive object creation and garbage collection cycles

To compare the mark, sweep and compact with the generational mark, sweep and compact algorithm, a test setup consisting of 5 consecutive object creation and garbage collection cycles was performed. In each cycle 500 objects with random references get created. Immediately afterwards, the heap is collected. Important in this test is, that before the next object creation phase starts, all references get erased. This results in leaving the surviving objects from the previous iteration as garbage for a short time, but in the object creation phase new references get created. Consequently, the mark, sweep and compact algorithm will erase a large percentage of the surviving objects from the last iteration, while the generational mark, sweep and compact algorithm leaves the survivors unexamined, as they are now in the old generation and will not be visited, until memory is completely exhausted.

The test parameters for this test were:

- 500 objects in the heap
- 5 % of all objects have root set membership, approximately
- 10 % of all objects have references to other objects, approximately
- 5 cycles of object creation with succeeding forced garbage collection
- 100 program runs

Figure 11 shows the total memory usage of the two algorithms. As expected, the maximum memory usage of the mark, sweep and compact algorithm stays on approximately the same level for each iteration. In contrast to this, the generational mark, sweep and compact algorithm memory usage slightly rises with each iteration, as garbage in the old iteration is not regarded. The generational algorithm however has less work to do in each iteration. The old generation will neither be traced nor collected, resulting in an approximately 50 ms faster runtime, as compared to the standard mark, sweep and compact algorithm runtime.

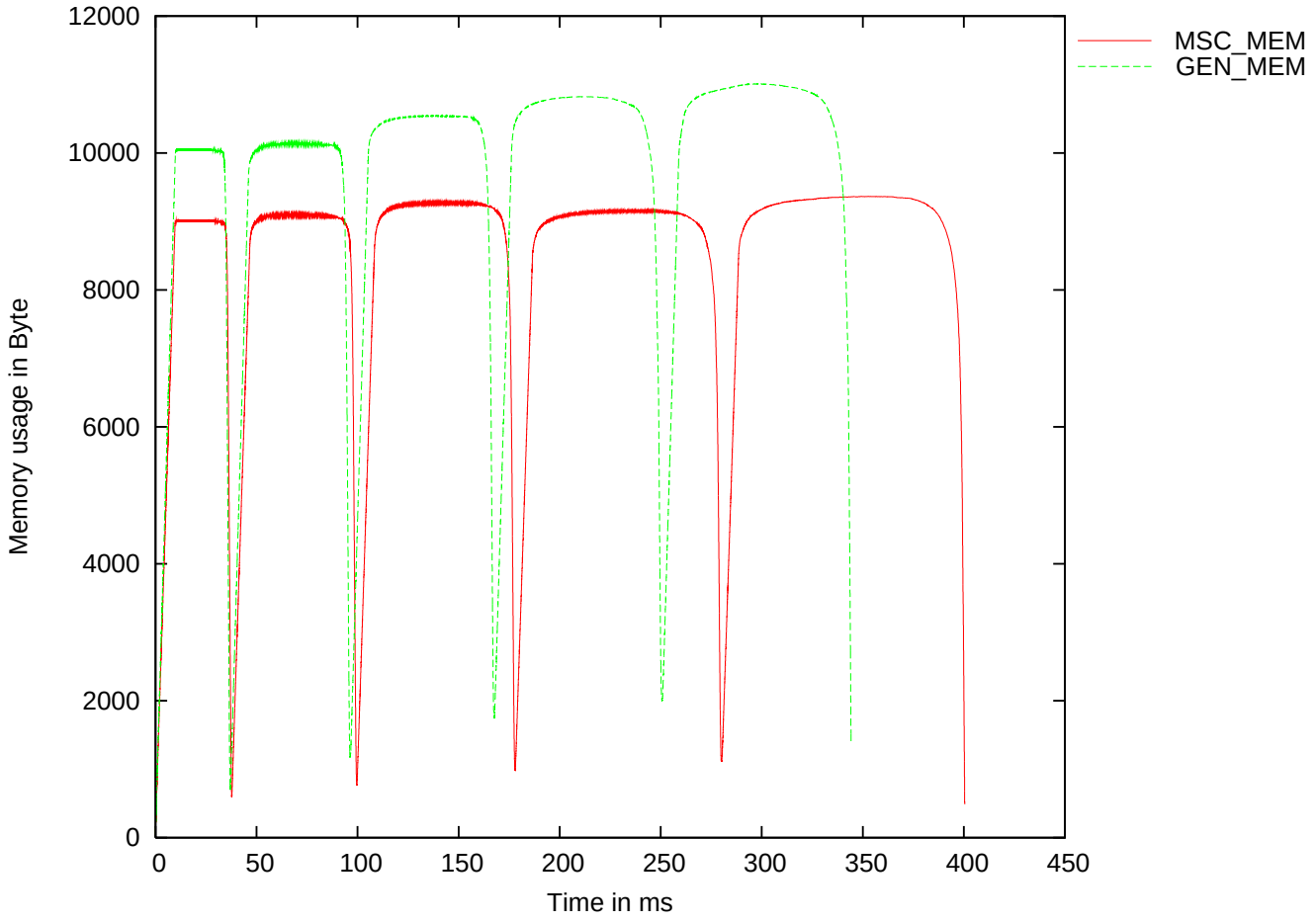


Figure 11: The total memory usage in test 3

6 Test review

In all the tests, the runtime of the in-place mark, sweep and compact algorithm was the longest as expected due to the higher complexity of the implementation. The generational algorithm had a sligthly worse runtime, when compared to the standard mark, sweep and compact algorithm except for test 4, where a better runtime is achieved by introducing a higher memory usage, because the algorithm has less objects to process. The stack usage of the two algorithms relying on recursion prove to be a significant factor and is not to be neglected. The reference count algorithm was generally the fastest because the reference count handling is done in C, but has the disadvantage of a fairly high memory overhead due to the reference count number carried with each object. Also, the unability of the collector to detect cyclic references is still an issue and the memory usage introduced by the standard mark, sweep and compact fallback algorithm has to be considered.

In general, the tests showed that a better runtime can be achieved if memory overhead is accepted. However, especially in a resource-constrained environment like sensor motes, the memory overhead leads to more frequent garbage collections, because the memory will exhaust faster. If memory overhead is undesired at all, the in-place mark, sweep and compact algorithm is the best choice, but trades off the least memory usage of all algorithms against a longer runtime.

The final test conclusion is that each of the proposed collector implementations can be used in a sensor mote environment. There is no clear best choice amongst the algorithms, as the selection of the used garbage collector is always a trade-off between runtime and memory usage and an algorithm fitting the current situation best has to be chosen.

7 Summary

The aim of this thesis was to propose Java garbage collectors, suitable for Java virtual machines that target small mobile devices with very limited memory resources. Over a period of three months, several different garbage collection schemes have been explored in theory and four different garbage collection algorithms have been implemented as part of the custom-built TakaTuka Java virtual machine. Besides confirming their correctness, the garbage collector implementations have been tested in different scenarios, with varying amounts of available memory, different levels of heap occupation and different reference density.

This thesis gives an overview of the meaning of garbage collection, the advantages and disadvantages of garbage collection and the most important traditional garbage collection techniques and variants. The results of the afore-mentioned tests have been presented and discussed to compare the different techniques especially with respect to usage in resource-constrained environments like sensor motes. Sensor mote applications are designed to be running for a long time and the correct reclaiming of unused memory is an important task. With garbage collectors such as the proposed ones, an easy way to guarantee that no physical memory leaks occur is achieved.

7.1 Weaknesses and future prospects

It has to be considered a weakness of this thesis, that the test scenarios that were provided are artificial in their nature and do not give consideration to a realistic sensor mote application to its fullest degree. A more meaningful test metric would be to test the algorithms with real life sensor mote applications.

Furthermore, the presented algorithms are fairly traditional in the way, that they are implemented. A more sophisticated approach, such as for example the implementation of a limited field reference count algorithm may lead to better results in practice. Other means of automatically triggering a garbage collection, like for example determining a suitable point in time to perform a garbage collection are left to be desired. Lastly, there is no mechanism to allocate objects in the middle of the heap address space. Even if a large percentage of the

heap is unoccupied, the heap could be considered “full”, because the pointer marking the point for succeeding allocations is at the very end of the heap space. The introduction of a bin-packing algorithm would improve this situations, as garbage collections would become less frequent.

A Garbage collector pseudocode

This appendix shows pseudocode for the different garbage collector implementations.

A.1 Implementation 1 - mark, sweep and compact

Pseudo-code of the marking phase:

```
while(rootSet has objects){
  traceRecursive(next object in the rootSet);
}

traceRecursive(object){
  if(object is black){
    return;             //object is already processed
  }

  set the object marking to grey;  //object is reachable

  for(each referenced object){
    if (referencedObject marking is white){
      set referencedObject marking to grey;  //set the referent to grey
        traceRecursive(referencedObject);    //recursive call
    }
  }
  //after returning from the recursion, mark the object black.
  //the object and all of its referents are now completely traced.
  set object marking to black;
}
```

Pseudo-code of the sweep-and-compact phase:

```
sweepAndCompact(){

  for(each object in the heap){
     if(object is not marked white){
        /*
         * CASE 1: RELOCATE MARKED OBJECT
         * slide the object to the lower end of the heap
         */
         if(there is a gap in the heap){
           slide the object to a lower position in the heap;
         }
         set the object marking to white;  //reset object marking
                                           //for succeeding runs.
      }else{
        /*
         * CASE 2: REMOVE UNMARKED OBJECT
         * delete the object, as it is garbage
         */
         delete the object;
      }
  }
}
```

A.2 Implementation 2 - in place mark, sweep and compact

Pseudo-code of the marking phase:

```
trace(){
  leftEnd, rightEnd;
  while(rootSet has objects) {
    mark object as encountered;
    if(object is outside the range of leftEnd or rightEnd){
      adjust leftEnd and rightEnd accordingly;
    }
  }
  while(true){
    for(each object between leftEnd and rightEnd){
      if(the object between leftEnd and rightEnd has been encountered){
        for(each referencedObject){
          if(referencedObject is unmarked){
            mark referencedObject as encountered;
            if(referencedObject is outside the range
                              of leftEnd or rightEnd){
              adjust leftEnd and rightEnd accordingly;
            }
          }
        }
        set the object marking to processed;
        if(no objects were marked in this iteration){
          break the while(true) loop;
        }
      }
    }
  }
}
```

A.3 Implementation 3 - generational mark, sweep and compact

Pseudo-code of the marking phase:

```
while(rootSet has objects){
  traceRecursive(next object in the rootSet);
}
if(collector is currently in minor collection mode){
  while (rememberedSet has object) {
    traceRecursive(next object in the rememberedSet);
    }
}

traceRecursive(object){
  if(object is black){
    return;             //object is already processed
  }else if(object is in the old generation &&
           collector is currently in minor collection mode){
    return;             //only the young objects are traced
  }
  set the object marking to grey;  //object is reachable

  for(each referenced object){
    if (referencedObject marking is white){
      set referencedObject marking to grey;  //set the referent to grey
        traceRecursive(referencedObject);    //recursive call
    }
  }
  //after returning from the recursion, mark the object black.
  //the object and all of its referents are now completely traced.
  set object marking to black;
}
```

Pseudo-code of the sweep-and-compact phase:

```
sweepAndCompact(){

  for(each object in the heap || each object in the young generatin){
     //depending on the collector mode (minor or major collection)
     //only the young generation or the whole heap is traversed

     if(object is not marked white){
        /*
         * CASE 1: RELOCATE MARKED OBJECT
         * slide the object to the lower end of the heap
         */
         if(there is a gap in the heap){
           slide the object to a lower position in the heap;
         }
         set the object marking to white;  //reset object marking
                                           //for succeeding runs.
      }else{
        /*
         * CASE 2: REMOVE UNMARKED OBJECT
         * delete the object, as it is garbage
         */
         delete the object;
      }
  }
}
```

A.4 Implementation 4 - reference count

Pseudo-code of the compact phase in the case of a minor collection:

```
compact(){
 for(each object in the heap){
   if(there is a gap in the heap){
     slide the object to a lower position in the heap;
   }
   set the object marking to white;  //reset object marking
                                     //for succeeding runs
 }
}
```

List of Figures

References

[1] J2me cldc and k virtual machine: Frequently asked questions. `http://java.sun.com/products/cldc/faqs.html`.

[2] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2004. ACM.

[3] D. L. Detlefs. *Concurrent atomic garbage collection*. PhD thesis, Pittsburgh, PA, USA, 1990. Chairman-Jeannette Wing.

[4] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[6] P. Griffin, W. Srisa-an, and J. M. Chang. An energy efficient garbage collector for java embedded devices. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 230–238, New York, NY, USA, 2005. ACM.

[7] R. Jones and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[8] M. Jump and K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. *SIGPLAN Not.*, 42(1):31–38, 2007.

[9] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, 1983.

[10] T. Lindholm and F. Yellin. *The Java^{TM}Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[11] J. Maebe, M. Ronsse, and K. DeBosschere. Precise detection of memory leaks. In R. Evans, D. Lencevicius, editor, *Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004)*, pages 25–31, Edinburgh, Schotland, 5 2004.

[12] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006. ACM.

[13] D. S. Wise1 and D. P. Friedman. The one-bit reference count. *BIT Numerical Mathematics*, 17(3):351–359, 9 1977.