Bachelor's thesis

Garbage collection for TakaTuka

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

Author: Luminous Fennell
matriculation number: 2111591

Supervising tutor:	M. Sc. Faisal Aslam
1st Examiner:	Prof. Dr. Christian Schindelhauer
2nd Examiner:	Prof. Dr. Peter Thieman

submission date: 11. May 2009

Abstract:

The *TakaTuka* project currently is developing at the Albert-Ludwigs University of Freiburg a Java virtual machine for different micro-controller architectures. It aims to enable programming of *sensor motes*, the nodes of a wireless ad-hoc network, by means of a high level language.

An important part of the Java programming language is its automatic memory management: dynamically allocated objects that are freed by the virtual machine when they are not used any more by the program it executes. This procedure is called Garbage-Collection. It is the only possibility in Java to free dynamic memory and therefore mandatory to reasonably execute applications.

The goal of this bachelor's thesis is the implementation of a functioning Garbage-Collection and an underlying, hardware independent memory management system for *TakaTuka*. This makes it possible for the first time to develop Java applications for *TakaTuka* that take advantage of the high level features of the Java programming language.

During the thesis several variants of memory management and Garbage-Collection were implemented. They are compared by means of test programs

Zusammenfassung:

Das *TakaTuka* Projekt entwickelt zur Zeit an der Albert-Ludwigs Universität Freiburg eine Virtuelle Maschine für Java, die es erlaubt Java Programme auf verschiedenen Microcontroller Architekturen auszuführen. Damit soll die Programmierung von Knoten drahtloser ad-hoc Netzwerke, sog. *Sensor Motes*, mit einer Hochsprache ermöglicht werden.

Ein wichtiger Bestandteil der Programmiersprache Java ist das automatische Speichermanagement: dynamisch allokierte Objekte werden selbständig von der Virtuellen Maschine freigegeben, wenn diese im Programm nicht mehr verwendet werden. Diesen Vorgang bezeichnet man als *Garbage-Collection*. In Java ist dies die einzige Möglichkeit dynamischen Speicher freizugeben und deshalb notwendig um sinnvoll Anwendungen auszuführen zu können.

Ziel dieser Bachelor Arbeit ist die Implementierung einer funktionsfähige Garbage-Collection für *TakaTuka* zusammen mit einem zu Grunde liegendem hardware-unabhängigen Speichermanagementsystems. Dadurch wird es erstmals möglich komplexere Java Anwendungen unter *TakaTuka* für die Sensor Motes zu entwickeln, die die Vorteile der Programmiersprache ausnutzen.

Im Laufe der Arbeit wurden verschiedene Varianten für Speichermanagment und Garbage-Collection implementiert, die anhand von Testprogrammen verglichen werden.

Acknowledgement:

I like to thank Faisal Aslam, Prof. Christian Schindelhauer and Prof. Peter Thiemann for their support and for giving me the opportunity to work on such an interesting project. I also would like to thank my family and especially my girlfriend for their patience and support during the time I spent on this thesis. I will make it up to you.

Declaration

I hereby declare that this thesis has been composed by me without any assistance and I have not used any sources or tools other than those cited. Furthermore I declare that this thesis has not been accepted in any other previous application for a degree.

Place, Date   Signature

Contents

1 Introduction

1.1 Background

Wireless Sensor Networks (WSN) are a very active research field that evolved recently. Their purpose is to monitor and control processes in an remote environment that is too vast or inaccessible to monitor with traditional centralized (wired) computer systems. WSNs consist of numerous tiny computer devices, so called sensor motes, which are connected via a wireless ad-hoc network. These sensor motes are placed inside the environment and should operate autonomously, ideally without the need of maintenance. They run on batteries for a long time and therefore have to operate very energy efficient. Because of this sensor motes typically have very limited computational power and memory resources compared to a standard PC.

Sensor motes are usually programmed using low level languages like C and nesC [6]. This inhibits development of new software for sensor motes as working with these languages is error prone and inconvenient: They are very hardware specific and don't support modern software engineering techniques and design patterns teached at universities. Therefore there exists a need for an implementation of standard compliant high level languages for sensor motes.

Java is a good example of such a language as it is hardware independent, very popular and has a vast standardized run-time library support. There exist even a dedicated library standard for devices with limited resources, the CLDC standard [16].

1.2 The *TakaTuka* Project and the *TakaTuka* Java-VM

The *TakaTuka* [2] project aims to provide Java for support several popular sensor motes, like the Crossbow Mica2 [18] mote. One part of the project is the development of a static class-file-loader that creates a optimized, statically linked binary file (called *tukfile*) from the class-files created by a standard compliant Java compiler.

The other part of the project is the development of a CLDC compliant Java-VM written in C. It runs as a TinyOS component. TinyOS is a programming framework built on nesC which supports multiple sensor motes. As the Java-VM can use the TinyOS drivers it can be easily adapted to support these different targets. The design goals for the *TakaTuka* Java-VM are

- low memory requirement to leave as much of the limited memory available to the Java program it executes,
- fast execution to save energy,

- hardware and library independence, so that *TakaTuka* can be maintained for multiple targets.

There are numerous efficient Java-VM implementations available but most of them target architectures with much more resources available than the typical sensor mote. Therefore a lot of existing optimization techniques [17] and strategies cannot be easily transferred to the *TakaTuka* Java-VM. For instance, the use of a *just-in-time* compiler (*JIT*) considerably speeds up the execution of bytecode compared to interpreting it. This approach requires the dynamic creation of machine code at run-time. Most sensor motes, however, have separated data and program memory (as defined by the *Havard Architecture*) and the latter is not reasonably accessible at run time. Other techniques as the use of caches or hash-tables are memory consuming (i.e. they enhance speed by using more memory) and too wasteful for a Java-VM for sensor motes.

1.3 Motivation for this Thesis

An important feature of the Java language is automatic memory management. The Java-VM has to provide a Garbage-Collection mechanism that reclaims dynamically allocated memory.

Prior to this thesis the *TakaTuka* Java-VM supported the execution of nearly all instructions of the hardware independent Java bytecode produced by a standard Java compiler. It also supported the Java multithreading facilities. However there was no automatic memory management implemented, which made development of realistic software for *TakaTuka* impossible.

Furthermore the memory needed by the Java program that the Java-VM executes was managed using standard library functions like the well known `malloc`. The implementation of the underlying memory management is was specific to the corresponding library and architecture. A reasonable Garbage-Collection implementation tightly integrates into memory management and therefore it cannot be library dependent considering the numerous target architectures that *TakaTuka* aims to support.

This thesis focuses on implementing a library independent memory management for the *TakaTuka* Java-VM including an automatic Garbage-Collector. The goals are to provide a stable and controlled way of organizing the limited memory of typical sensor motes while trying to keep computational and memory overhead small.

1.4 Related Work

There exists already a bachelor's thesis on Garbage-Collection for sensor motes and the *TakaTuka* project [14]. It was implemented mainly in Java code using an interface of native methods to access the memory. Although promising, this approach turned out to be to inefficient for a reasonable use in the *TakaTuka* Java-VM. Also the Java-VM, and therefore the also Garbage-Collection implementation did not support threading at the time. The implementation discussed in this thesis is written in C and integrated into the *TakaTuka* Java-VM.

There exists also another Java-VM that aims to support the same types of architectures as *TakaTuka* [4]. At the time of writing it has Garbage-Collection already implemented but only supports a subset of the Java language.

1.5 Structure of the Thesis

The rest of the thesis is structured as follows: The second section gives an overview of the general run-time memory usage of a Java-VM. The third section will give a general overview over different Garbage-Collector types and a justification of the choice made for *TakaTuka*. Details about the implementation of dynamic memory management in *TakaTuka* are given in the fourth and fifth section. Also the actual Garbage-Collection scheme is described. The sixth section covers results of the comparison of different memory management schemes in terms of computational effort and memory overhead. Finally a summary and motivation for future work is given in the seventh section.

2 The Java Programming Language

In this section technical aspects of the Java programming language and Java-VMs are reviewed as far as relevant for memory management. For further details on the Java language see [1].

2.1 The Benefits of the Java Programming Language

Java is a modern, object oriented, high level language. It is statically typed and based on the C syntax but provides language support for classes and class hierarchies. Java is designed to be highly portable, hardware independent and safe. It therefore, in contrast to C++, prohibits direct memory access and pointers and features automatic memory management and reference semantics for class instances.

Java programs are compiled into portable standardized bytecode instructions that are execute by a by a virtual machine (Java-VM). The functionality of a Java-VM is defined by the *Java Virtual Machine Specification* [12] which ensures a unified behavior of Java programs across all platforms.

Because of its portability and simple design, Java is easy to learn and widely used. It therefore is an ideal candidate for a high level language for sensor motes as users new to the field are not forced to learn an unfamiliar and error prone low level language like C or nesC. The development of prototypes and test programs can be accelerated if the programmer can focus on the algorithms and is not forced to deal with low level concepts as manual memory management.

2.2 Java Bytecode and Java Virtual Machines

The Java compiler translates Java source code into so called *class-files.* The format of the class-files is standardized in the Java Virtual Machine Specification and contains mainly:

- class definitions: type and name of instance variables, class hierarchy, etc...
- method definitions: types of arguments and local variables, operand stack size (see below), and a sequence of virtual machine instructions called *bytecode*

A Java-VM is a program which reads class-files and executes the contained Java program. The Java-VM essentially translates the semantics of the bytecode instructions to the target architecture on which it is run. This can be done by interpretation of the bytecode instructions at run-time or compilation into native machine code or a combination of both.

The bytecode operates on values of numeric types (corresponding to numeric types in the Java language; `boolean, byte, short, int, float, double`) and of a special type called `reference`. The values of type `reference` identify object instances and array instances which are stored on the heap (see below). Furthermore the Java bytecode is *stack based.* Instead of manipulating the content of registers as most machine code instruction do, the Java bytecode instructions uses a so called *operand stack* for its operations. A commented example is shown in Figure 1.

For the

invocation of each method such an operand stack is created. The values that are obtained by method calls as the result of calculations are pushed onto the operand stack of the currently executing method. They then can be either stored in variables (static or local) or instance variables of class instances and fields of arrays respectively. Also the arguments for method calls and the operands of logical and arithmetic operations and control flow instructions (i.e. conditional jumps) are taken from the operand stack.

As the types of return values of methods, the types of local variables and of instance variables are known from the class-file, it's possible to reconstruct the types of the values on the operand stack during execution of the bytecode. It's even possible to reconstruct the types of the values on the operand stack at every instruction at compile time which is used for *bytecode verification* (see [12] for details).

The size of the operand stack of a method has an upper bound which is calculated at compile time and stored in the class-file.

```
/* simply adds 3 to its argument */
public static int add3( int i ){
    return i+3;
}
public static int m1( int a, int b ){
        int c = add3(a);
        return c-b;
}
```

bytecode	pseudo code	op	v
	#start execution	[3]	[3,2,-]
iload_0	op.push(v[0])	[3]	[3,2,-]
invocestatic #2	tmp = op.pop()	[]	[3,2,-]
	tmp = add3(tmp)	[]	[3,2,-]
	op.push(tmp)	[6]	[3,2,-]
istore_2	v[2] = op.pop()	[]	[3,2,6]
iload_2	op.push(v[2])	[6]	[3,2,6]
iload_1	op.push(v[1])	[6,2]	[3,2,-]
isub	tmp1 = op.pop()	[6]	[3,2,-]
	tmp2 = op.pop()	[]	[3,2,-]
	op.push(tmp2 - tmp1))	[4]	[3,2,-]
ireturn	return_method(op.pop())	-	-

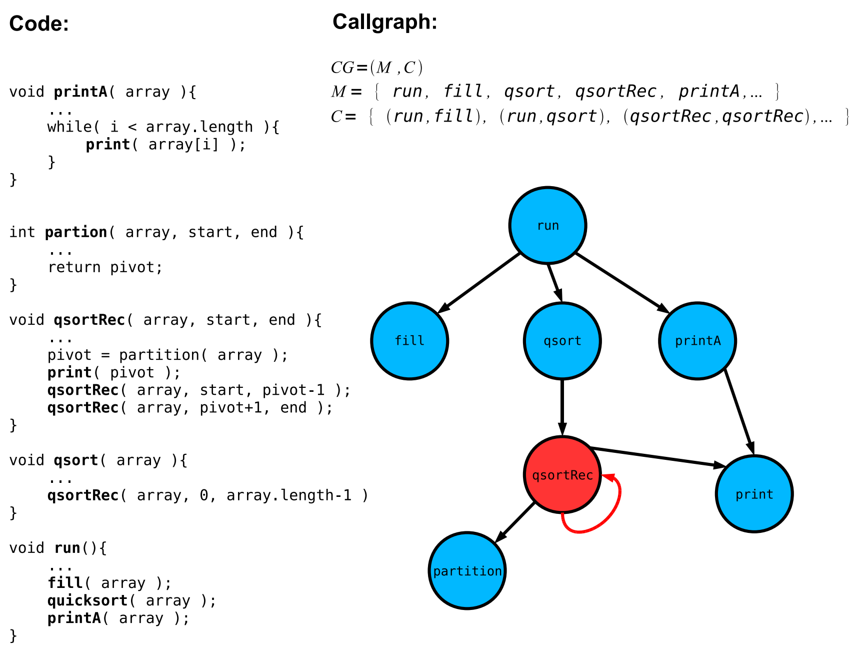
Figure 1: Java bytecode of method `m1` is shown in the table. The explaining pseudo code uses an operand stack `op` and an array of local variables `v`. The example values of operand stack and variables are for the method call `m1( 3,2 )`.

2.3 Execution of a Java Program

Execution of a Java program starts with the `main`-method from which other methods are called. As in any procedural programming language, if a method is called the execution of the current method is stopped until the called method returns. This ensures that all code is executed sequentially: the `main` method starts a single *thread of execution* or *thread*.

It's possible to introduce additional threads in Java by creating an instance of a subclass of the `Thread` class and calling it's inherited `start` method on the class instance. Then the thread becomes *active* and the code specified in the `run` method of this class is run in parallel to the other threads. The instance of the `Thread` class will be called *thread object* in the following. It will be important in Section 2.4.2. If the `run` method of a thread object returns the thread will become inactive.

2.3.1 Call Graph and Call Chain



Code:

```
void printA( array ){
    ...
    while( i < array.length ){
        print( array[i] );
    }
}

int partion( array, start, end ){
    ...
    return pivot;
}

void qsortRec( array, start, end ){
    ...
    pivot = partition( array );
    print( pivot );
    qsortRec( array, start, pivot-1 );
    qsortRec( array, pivot+1, end );
}

void qsort( array ){
    ...
    qsortRec( array, 0, array.length-1 )
}

void run(){
    ...
    fill( array );
    quicksort( array );
    printA( array );
}
```

Callgraph:

$CG=(M,C)$

$M=\{$ *run*, *fill*, *qsort*, *qsortRec*, *printA*,... $\}$

$C=\{$ (*run*,*fill*), (*run*,*qsort*), (*qsortRec*,*qsortRec*),... $\}$

Figure 2: Example of cyclic call graph

The structure of method calls in a thread is given by the *call graph*

Definition 2.1 (Call Grap)**.**
A *call graph* is a directed graph

$$CG = (M, C)$$

with nodes

$$M \subseteq \{\text{compiled Java methods}\}$$

and edges

$$C = \{(m_1, m_2) | m_1, m_2 \in M \wedge \ m_1 \text{ calls } m_2 \ \}$$

An example of a call graph is given in Figure 2. A path in the call graph, a sequence of method calls, is called a *call chain*:

Definition 2.2 (Call chain)**.**
A *call chain* of a call graph $CG = (M, C)$ is a finite sequence of methods

$$CC = (m_1, \ldots, m_n)$$

with

$$m_n, m_i \in M \textbf{ and } (m_i, m_{i+1}) \in C, \ 1 \leq i < n$$

A call chain has a length defined as

$$\mathit{length}(CC) = n$$

Definition 2.3 (Recursive Thread)**.**
A thread is *recursive* iff its call graph $CG = (M, C)$ is cyclic, i.e. a call chain $CC = (m_1, \ldots, m_n)$ exists with

$$m_i = m_j \textbf{ and } i \neq j, \quad i, j \in \{1, \ldots, n\}$$

The call graph of a thread that is not recursive is a tree. The root of this tree is either the `main` method or the `run` method of the thread. Special call chains can be defined for this call graph:

Definition 2.4 (Maximal Call Chain)**.** A maximal call chain $CC_{max} = (m_1, \ldots, m_n)$ of a acyclic call graph CG is a call chain for which m_1 is the root of the tree and m_n is a leaf:

$$\neg\exists m_i \in M : (m_i, m_1) \in C \textbf{ and } \neg\exists m_i \in M : (m_n, m_i)$$

2.4 Memory Usage in Java

A Java-VM has to provide and manage the memory needed by the Java program it is executing. This task can be divided in management of *stack memory* and *heap memory.*

2.4.1 Stack Memory

For each methods that is called data like local variables and the current state of the operand stack has to be stored until the method returns. For this a data record called *stack frame* is used:

Definition 2.5 (Stack Frame)**.** A *stack frame* is a data record associated with a method. It contains:

- *method-id:* a value identifying the method associated with the frame,
- *program-counter:* a pointer to the next instruction to be executed in the bytecode of the associated method,
- *local-variable-buffer:* a fixed sized array holding the local variables of the associated method,
- *operand-stack-buffer:* a fixed sized array designated to hold the values of the operand stack. The operand stack itself grows and shrinks inside this array. The operand-stack-buffer is sized to contain the maximum size of the operand stack. This size can be determined at compile time (see above)
- *frame-pointer:* an indicator to the current top value of the operand stack

Definition 2.6 (Method Stack)**.** A *method stack* is a dynamically sized data-structure which implements a stack of (differently sized) stack frames.

A method stack represents the state of a single thread. When a method is called (through a bytecode instruction) a new stack frame for the called method is created and pushed on top of the method stack. When the methods returns the stack frame is popped from the method stack.

2.4.2 Heap Memory

One of the main differences of Java compared to low-level languages like C/C++ is that the programmer doesn't have direct access to the physical memory used by the Java program. Instead dynamic memory is accessed through *references* (values of type `reference`) which identify class instances and array instances. In the following all instances (of a class or an array) will be referred to as *object*. Unlike pointers (e.g. in the C programming language [10]) *references* don't (necessarily) indicate an address in memory. There is no equivalent of *pointer arithmetic* in Java and no possibility to manipulate the memory associated with the reference value other than through access of *instance variables* of the corresponding class instance or of array fields. Therefore it's not possible for the programmer to move, delete or overwrite arbitrary memory regions.

New references are created by the `new` instruction. This instruction specifies either a class to be instantiated or a type for an array and its length. The Java-VM is responsible to provide the memory needed by the corresponding object and to associate a reference with it. In the case of class instances the size of the region is known at compile time (defined by the class definition). The length of array instances can be run-time dependent. The memory region where the memory of all instances is located is called the *heap*.

Objects that were instantiated can only be accessed through their reference. At any time during the execution of the Java program, a special set of objects can be defined.

Definition 2.7 (Root Set)**.** The *root set* at a certain time during the execution of a Java program is the set of objects that can be referenced without accessing other objects. An object o with reference r is part of the root set iff it satisfies at least one of the following conditions:

- any static variable of type `reference` has the value r,
- any instance variable of type `reference` of a thread-object that is active has the value r,
- r is the value of at least one local variable in any of the stack frames of the method stacks

- r is at leads once on the operand stack of any of the stack frames of the method stacks

Objects that are not in the root set can only be accessed through references that are stored in other objects. The root set is the "starting point" to access these objects.

Definition 2.8 (Referencing Objects)**.** An object o_1 *references* an object o_2 iff o_1 has an instance variable or array field of type `reference` that has the value of the reference of o_2.

Definition 2.9 (Reachable Objects)**.** An object o is *reachable* iff

- it is part of the root set
- or it is referenced by a reachable object.

It is obvious from the above definition that the data contained by unreachable objects can never be accessed by the program in the future. The Java-VM can now re-associate the memory region to a different object and also can reuse the reference.

The reachable objects form a *reference graph.*

Definition 2.10 (Reference Graph)**.** A *reference graph* is a directed graph

$$RG = (O_r, R)$$

with nodes

$$O_r := \{\text{reachable objects}\}$$

and edges

$$R := \{(o_1, o_2) | o_1, o_2 \in O_r \wedge o_1 \text{ refers } o_2\}$$

3 Garbage-Collection for a Java-VM for Sensor Motes

In the present context Garbage-Collection means that the Java-VM has to identify the memory occupied unreachable objects can be identified and *freed* (meaning returning it to a state where it can be reassigned to other objects). The Java language doesn't provide the possibility to free objects manually so a Java-VM needs to provide automatic Garbage-Collection. The part of the Java-VM which is responsible for this task is called the Garbage-Collector.

3.1 Constraints for the Garbage-Collector

There exist different approaches for Garbage-Collection (see [9] for an overview). When implementing a Garbage-Collector for a Java-VM for sensor motes one should consider certain constraints.

- Typically Garbage-Collection techniques introduce a memory and run-time overhead to the memory management. This gets important when dealing with motes that have very little memory and computational power. This overhead should be kept as low as possible.
- The programs run by individual sensor motes are not usually critical applications, as sensor motes are designed to be placed in an non controlled environment and therefore are expendable. Enforcing real time constraints on the Garbage-Collector therefore could be unnecessary.
- The typical run-time behavior of a sensor mote consists of rather long periods of inactivity interrupted by shorter periods, when an event triggered a sensor, where information has to be processed and data transmitted. The active periods should be interrupted as little as possible.
- The properties of the Java language and the typical Java programming style imply a heavy use on *temporary objects*. For example the string literal in Java is implicitly an instance of class `String` and therefore placed on the heap. It is also a common practice for Java programmers to create iterators which are small objects only used for the execution of a loop and never used again afterwards. It would be advantageous if the Garbage-Collector could detect and free those temporary objects efficiently.

3.2 The Mark-Sweep Garbage-Collector

The Garbage-Collector approach chosen as the base of Garbage-Collection for *TakaTuka* is the so called *mark-sweep* Garbage-Collector [13]. First the basic functionality of this approach will be briefly described. The actual implementation is described in more detail in Section 4.5. Reasons why the two other classical approaches (*reference counting, copying Garbage-Collector* were not chosen are given afterwards.

The basic algorithm is given in Algorithm 1. In a first phase (called the *mark-phase*), all reachable objects are identified. This is done analogous to Definition 2.9 by marking all objects in the root set. During the marking of an objects all objects refered by it (through instance variables or array fields of type `reference`) are also marked recursively. In the second phase (*sweep-phase*) the heap is scanned for unmarked objects (by definition not reachable) which can be freed.

Algorithm 1 Mark-sweep Garbage-Collector

```
#this function frees an object
function free( o )

#this function returns the object
#belonging to reference 'ref'
function lookup_object( ref )

function mark_object( o ):
    if o.marked == true:
        return
    o.marked = true
    for f in instance variables resp. array fields of o:
        if f is a reference:
            mark_object( lookup_object( f ) )

function mark():
    for v in static variables:
        if v is a reference:
            mark_object( lookup_object( v ) )
    for t in active threads:
        mark_object( t )
        for v in method stack of t:
            if v is a reference:
                mark_object( lookup_object( v ) )

function sweep():
    for o in all objects:
        if o.marked == false:
            free( o )

function gc():
    mark()
    sweep()
```

In this form the Mark-Sweep Garbage-Collector follows a *stop-the-world* approach: the interpretation of bytecode is stopped during the invocation of the Garbage-Collector. In the mark-phase all reachable objects have to be searched and in the sweep-phase all objects have to be checked for marks. Therefore it could take a considerable amount of time until the interpretation of bytecode can continue. This drawback could be acceptable for typical sensor mote applications as the Garbage-Collector could be run at the beginning of a phase of inactivity. However this would require that there is enough memory available to complete a phase of activity. This is less likely as Java programs tend to create many temporary objects which will fill up the heap very quickly. In the next section an approach which could act as support to the mark-sweep Garbage-Collector is discussed which can deal with these objects.

3.3 Escape Analysis

To avoid that a stop-the-world Garbage-Collection would be triggered very often because of temporary objects, it could be beneficial to try to identify these objects at compile time by statically analyzing the bytecode.

```
 1  class SomeClass{
 2      private Vector warnings;
 3
 4      public int someMethod( int loopEnd, int check ){
 5          int result;
 6          for( int i = 0; i < loopEnd; i++ ){
 7              result += new Processor( i ).process();
 8          }
 9          /* from here on, no processor object
10           * is used anymore. All objects created
11           * by the new-instuction in line 7 could be
12           * free'd */
13          if( result > check ){
14              this.warnings.add( new Warning( result )
    );
15          }
16          return result;
17      }
18  }
```

Figure 3: Escape analysis example

Consider the Java code in Figure 3. The objects created in the loop in line 7 obviously are never used outside of this loop. Their references are even never stored in a variable. These objects are *local* to the method and there is no reason to keep their memory after the method returns. In contrast, the reference of the object created in line 10 is stored inside a heap allocated vector. This reference *escapes* the method and the lifetime of the object can not be determined by examining the code.

The procedure of determining if references of objects escape the local scope of a method or not is called *escape analysis*. It was first used to support Garbage-Collection in functional languages and then also in object oriented languages and Java [5]. In most approaches, the objects that do not escape the method are stored in the method's stack frame, e.g. [7] and therefore automatically freed when the method returns.

Escape analysis as it is planed for *TakaTuka* does not take this approach in order to keep the stack frame small. Reasons for this are given in Section 5. Instead objects are explicitly freed by a special bytecode instruction: objects that do not escape a method can be grouped by the specific `new`-instruction that is used for their creation. A new bytecode instruction, `gc`, is introduced which instructs the Java-VM to free a group of objects. These `gc` instructions are placed inside the bytecode by the class-file-loader (see Section 1.2). In the example in Figure 3 such an instruction could be placed after line 8.

Escape analysis functionality was not yet integrated into the *TakaTuka* Java-VM during this thesis as the basic facilities for memory management needed to be established first. However it is mainly orthogonal to the mark-sweep Garbage-Collector and could compensate for its above mentioned drawbacks when dealing with temporary objects.

3.4 Other Approaches for Garbage-Collection

There are other approaches and variants of Garbage-Collection than the Mark-Sweep Garbage-Collector. They will be described very briefly here as well as the reason they weren't chosen for the Garbage-Collector implementation in this thesis. For more detailed description see [9].

3.4.1 Reference Counting

A very intuitive approach for Garbage-Collector is *reference counting*: Every object gets a reference count initialized to 1 on creation. When the reference of the object is copied then the reference count is incremented, when a reference of the object is overwritten then the reference count is decremented. If the reference count reaches zero, the object is freed and the reference count of objects refered by the freed object is decremented. In this way objects are freed as soon as the last reference to them is overwritten. Although this would be favorable to the many temporary objects created in a typical Java program, this approach wasn't chosen because of the following reasons:

- Cyclic data-structures (like e.g doubly linked lists) never get deleted by reference counting because they always refer each other. As the usage of such data structures cannot be excluded (especially given Java's extensive standard library) supporting techniques would have to be introduced additionally to reference counting.

- Copying and assigning references has to be treated specially with this approach and introduces a computational overhead to these operations. As the Java-VM has a stack based execution model these operations are very common (e.g. method calls and instance variable access use each multiple assignments of references) and would probably slow down the execution of the bytecode considerably (even if only little dynamic memory is needed).

3.4.2 Copying Garbage-Collector

With this approach the heap is divided in two equally sized partitions. One of them is the active partition, the other the free partition. Memory is allocated consecutively in the active partition until Garbage-Collection is invoked (e.g. when there is no more memory available). Then the reachable memory is copied to the free partition. It can be marked similarly as for the mark-sweep Garbage-Collector. After this the free partition becomes the active one and vice-versa. The advantage of the Copying Garbage-Collector is that heap memory does not get fragmented (see below Section 4.3) when freeing objects but for sensor motes its not affordable to constantly waste half of the available memory. Also this approach implies movement of objects in the heap which entails updating their references (see Section 4.3.3).

4 Heap Memory Management for *TakaTuka*

Management of dynamic memory was reimplemented from scratch for the Java-VM of the *TakaTuka* project during this thesis. It was done in a architecture and library independent way to ensure reliable operation throughout the various target architectures *TakaTuka* aims to support.

This section deals with the issues emerging with such a task and the solution chosen for *TakaTuka*. First the memory layout used by the Java-VM is described. Afterwards the notion of allocating and freeing memory blocks of variable size is explained. Then measures to deal with fragmentation are described and finally the implementation of the Garbage-Collector and its integration into the allocation process is examined.

4.1 Memory available to the Java-VM

A C program uses three kinds of data memory. They typically all reside in the same physical memory (e.g. SRAM in the case of some sensor motes):

- Global variables map to *static memory*. These variables are mapped to memory by the C compiler at compile time and never change their address

- The local variables used during the execution of Java-VM code are mapped to *C-stack memory*. They exist only for the execution of a function and their memory is assigned by the C compiler. If no recursion is used for implementing the Java-VM, the amount of C-stack memory used during execution can be calculated at compile time.

- The rest of the memory is called *dynamic memory* and is accessed through local or global pointer variables which hold memory addresses. The run-time-library typically provides facilities to register memory for different data records so that they don't overlap with each other, the C-stack memory and static memory respectively. This is done for instance by calling special library functions like the `malloc` function of the C standard library.

The management of dynamic memory is dependent on the implementation of the standard library for the target architecture. The *TakaTuka* Java-VM therefore uses a large, fixed sized memory region in static memory for its run-time dependent memory requirements instead. It manages the usage of this region independently from any library functions and no further dynamic memory is used. Therefore *dynamic memory* denotes this fixed region in the following.

No recursion is used in the Java-VM implementation. The maximum amount of C-stack memory can be estimated at compile time and the size of the self-managed dynamic memory can be chosen accordingly.

4.2 Allocating and Freeing Memory Blocks

As mentioned above, the Java-VM has to manage the heap memory and has to provide contiguous memory regions of arbitrary size requested by the Java program. It has to transparently keep track of the requested regions. The approach chosen is straight-forward and resembles typical `malloc` implementations for micro-controllers, e.g. [3]. However it provides means for iterating through the allocated blocks which is necessary for Garbage-Collection.

4.2.1 Definitions

A continuous region of the dynamic memory is used to hold the objects (as defined in Section 2.4.2) of a Java program. It is called *heap memory.* It starts at the highest address of the dynamic memory and its end is stored in the pointer `heap-end`.

Definition 4.1 (allocating objects)**.** When a java program instantiates an object (using the `new`-instruction) the Java-VM has to return a reference which identifies a continuous memory region in heap memory. After this process the object is *allocated.* The Java-VM also has to ensure that the memory region of an allocated object is only modified through the corresponding reference.

Definition 4.2 (freeing objects)**.** Freeing an object denotes the process of invalidating its reference and preparing its memory region to be reused by the Java-VM. Typically an object is freed by the Garbage-Collector as the programmer has no means to free an object explicitly in Java.

In order to identify the memory regions of allocated objects each region is contained in a *memory block.*

Definition 4.3 (Memory Block)**.** A memory block is a variable sized data record in heap memory which consists of three parts:

- the `block-header` which stores an identification of the memory block (called its *id*) and its size,
- the `block-data` which is the memory region used by the Java program, and
- the `block-footer` which also stores the size of the memory block

It is to be noted that `block-header` and `block-footer` have fixed size while `block-data` is of variable size, depending on the size of the object requested by the Java program.

4.2.2 Example

The basic process of allocating and freeing blocks of memory of a certain size is now described by following the example given in Figure 4.

1. In Figure 4A the state of the heap after initialization is shown. It is empty so `heap-end` points to the start of heap memory.
2. In Figure 4B the Java program has requested several allocations of objects (through the `new`-instruction). The heap memory is increased to hold the allocated memory blocks by moving the `heap-end` pointer upward. The allocated objects are identified by the `id` of their memory block. This is also the reference returned to the Java program.

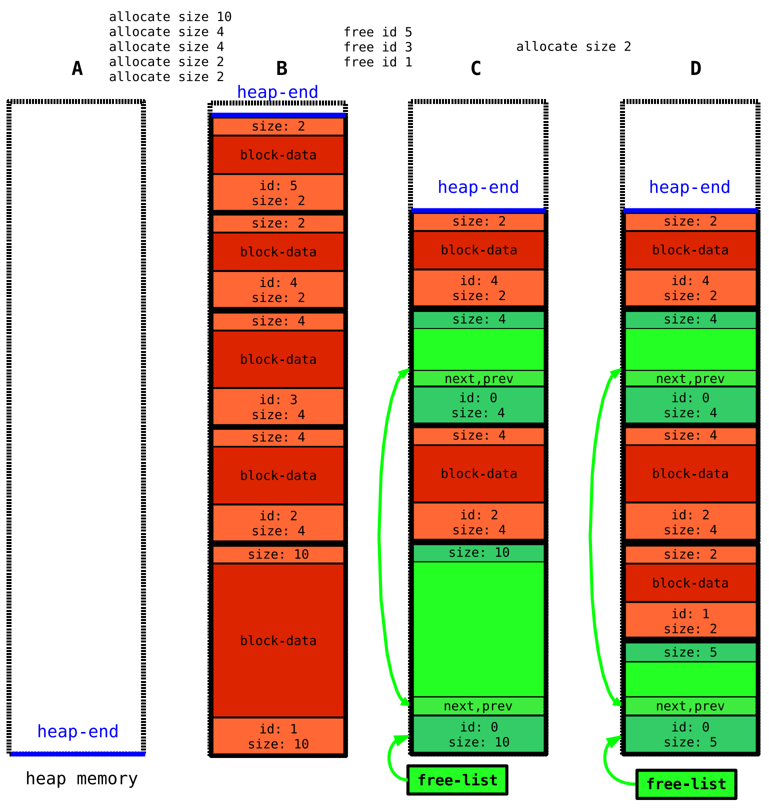


Figure 4: Allocating and freeing memory blocks

3. In Figure 4C objects have been freed by automatic Garbage-Collection. If a freed block is at the end of the heap memory, `heap-end` is moved down and the heap memory shrinks. Otherwise the block is marked as free by setting its `id` to a reserved value `null`. The block is then inserted into the so-called free list. This is a doubly-linked list of free memory blocks. The pointers which link the free list items are stored inside `block-data`, previously used by the object, to avoid adding to the overhead of a memory block.

4. For further allocations of objects (Figure 4D), the free list is searched for a free block of the requested or greater size. Currently a *first-fit* policy is implemented, where the list is searched linearly from the start until a matching free block is found. If this free block is of greater size the new block is allocated inside the free block and a new free list item for rest of the memory is created.

4.2.3 Properties of Allocating and Freeing

The management of heap memory has several properties:

- If the address of a memory block is known it's possible to jump to adjacent blocks. The next block (further up in heap memory) can be accessed because the size of `memory-data` is stored in the `block-header`. The previous block can be accessed because the size of the previous block is stored directly beneath the `block-header` of the current block.

- All memory blocks that are free have an `id` value of `null` and can thus be identified as free but not differentiated from each other. They only can be accessed by the free-list

- Allocation of objects generally implies searching the free list for matching free blocks. This is a computational overhead to allocation. Freeing an object on the other hand can be done quickly by inserting it into the free list.

- When a new object is allocated inside of a free block which is larger, another free block is created for the memory unused by the new object. This however is only possible if the unused memory can contain a new memory block and the pointers to insert it into the free list. If this is not the case then the size of the allocated memory region is increased and no free block is created.

- As can be seen in Figure 4D the heap memory becomes *fragmented* with time. The next section will deal with this issue.

4.3 Fragmentation

4.3.1 Definition

Fragmentation denotes the amount of separation of a memory region into used blocks and free blocks. With the previously introduced memory management strategy this separation occurs when inserting a memory block into the free list (see Figure 4C-D). A measurement for fragmentation could be

$$\text{Frag} := \frac{\text{number of free blocks}}{\text{amount of free memory in free list}}$$

A high fragmentation therefore means that the memory available in the free list is split up into many free blocks. This can be disadvantageous for further allocations. A moderately large object could not fit in any of the free blocks, even if the overall memory available in the free list would easily allow this. The heap memory would have to be increased in this case. Assume for instance that an object of size 6 is allocated in the situation of Figure 4D. This would lead to the described scenario.

Fragmentation is an intrinsic property of dynamic memory management of variable sized memory blocks. The allocation strategy described above increases fragmentation over time because the free blocks are split up during allocation leading to the same amount of free blocks for less free memory in the free list. As memory is very limited in sensor motes there should be taken measures to avoid fragmentation and increasing of the heap memory.

4.3.2 Combining Adjacent Free Blocks

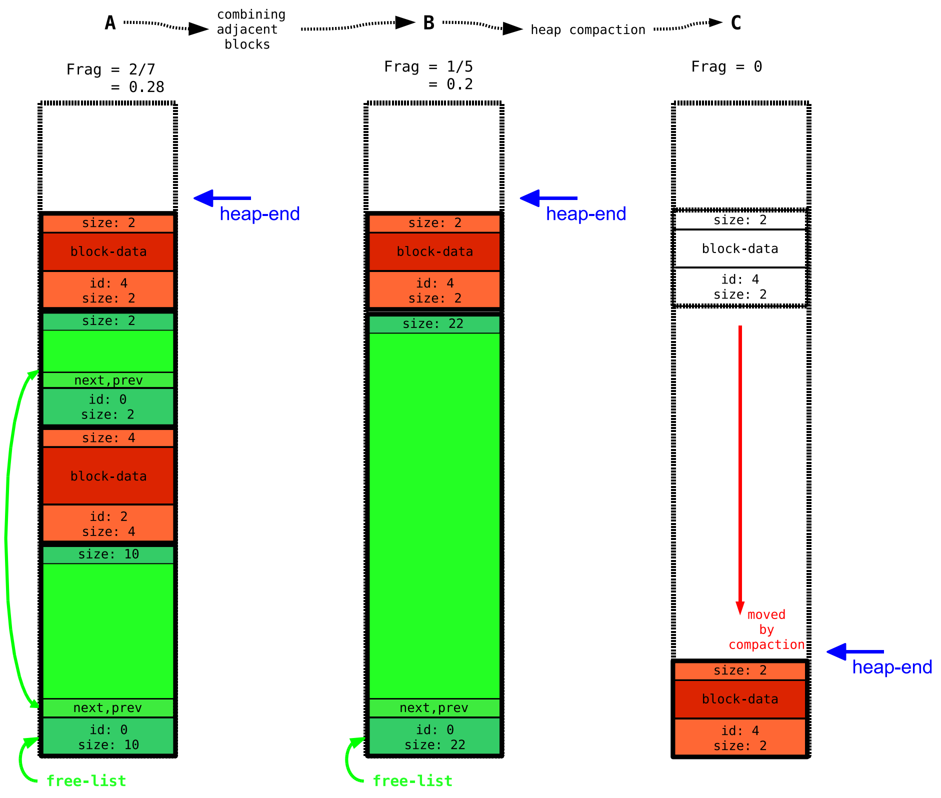


Figure 5: Combining adjacent block on allocation

When adjacent free blocks are combined fragmentation is reduced. This is because the number of blocks decreases. The amount of available memory in the free list even increases because every `block-header` except for one can be omitted and made available for `block-data`. As it is possible to identify adjacent blocks and if they are free (see Section 4.2.3) this can be done during insertion of a memory block in the free list (i.e. while freeing an object). See Figure 5 for an example. This strategy ensures that there are never two adjacent free blocks.

4.3.3 Heap Compaction

The fragmentation of a memory region can be reversed by relocating the used memory blocks so that there are no more free blocks between them. This process is called *heap compaction.* See Figure 5 for an illustration of this process and Appendix A for the corresponding algorithm.

However, if the references used to identify a memory block are dependent on the location of the block (as it's the case when using memory addresses as references for instance) then all references to the block become invalid when it's moved. This would mean that all of those references have to be updated according to the new location. This is a considerable effort because the instance variables of all reachable objects and all method stacks have to be searched for references to a moved block. Without the use of special data structures which could provide efficient identification of invalid references (by means of a balanced search tree for instance), this procedure has to repeated for every moved block. The use of those data structures would require additional dynamic memory which would have to be taken into account by the memory management.

4.4 Indirect References and Reference Table

The invalidation of reference at heap compaction can be avoided by using *indirect references* for accessing objects. With this approach the references of an object are independent from the position of its memory block. In the implementation for *TakaTuka* a *reference table* is maintained which maps the references to the corresponding addresses of their respective memory blocks. The references are simply the index to the position of this address in the table.

Figure 6 shows the difference between direct and indirect references when blocks are moved. In case of direct references, many references become invalid. They would have to be searched for and updated. If a reference table is used than only one direct reference, the one contained in the reference table, becomes invalid. The `block-header` as defined in Section 4.2.1 contains an `id` entry which is an indirect reference to the object itself. Through this reference the procedure performing the move of the block can update the single invalid direct reference in the reference table.

If a memory block is freed the unused reference table entry is inserted into the *reference table free list* by changing its value to the index of the next free table entry as shown in Figure 7. In contrast to the heap free list a unused reference table entry can be found by just taking the first item out of the free list as all entries are of equal size.

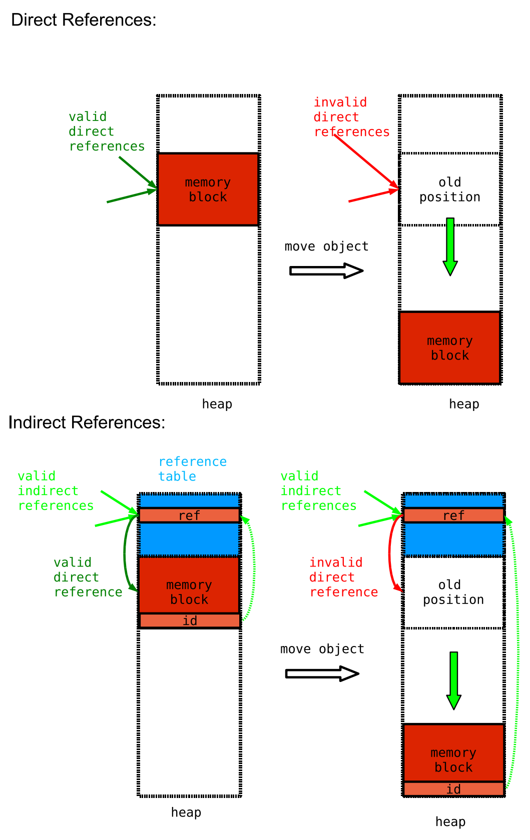


Figure 6: Direct and indirect references

4.4.1 Sizing of Reference Table

If the reference table is given a fixed size it limits the number of objects that can be used by the Java program simultaneously.An object that does not fit inside the table could not be refered. In other words the reference table has to be big enough to contain the references of all reachable objects at any time of the execution of the program. The maximum number of reachable objects would have to be predicted to determine the minimum size of the reference table. Also it is likely that the program uses less objects most of the time. In this case the reference table would waste memory in addition to the overhead it already introduces.

In order to counteract an over estimation of reference table size it is possible to allow the reference table to grow if more objects are needed than estimated. Therefore it has to reside in dynamic memory. It is initially placed at a fixed distance of the heap memory (e.g. the other end of the dynamic memory) and grows in opposite direction. It is to be noted, that the table can only shrink if its last entry is freed. In the example in Figure 7, for instance, the table cannot be shrinked because the indirect references

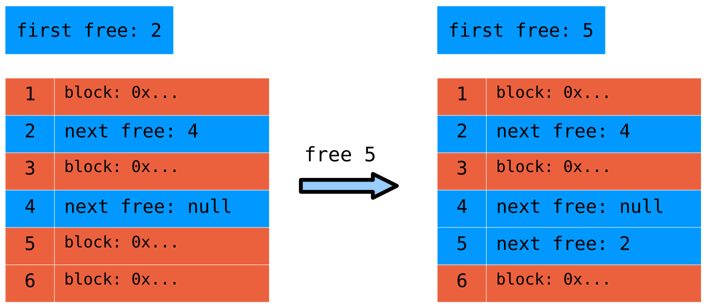


Figure 7: Reference table free list

pointing to object 6 rely on finding its direct reference (i.e. its memory address) at offset 6 in the table.

The reference table can also be moved if necessarily, as the references are relative to the start of the table. This is needed if heap memory has to share dynamic memory with other management processes, e.g. the stack memory management, see Section 5.

4.5 Mark-Sweep Garbage-Collector for *TakaTuka*

As the basic facilities for allocating and freeing memory blocks are now provided a Garbage-Collector has to be integrated into the allocation process. If the Java-VM detects that the Java Program runs out of memory the Garbage-Collector has to be invoked.

In Section 4.1 it was explained that the Java-VM should avoid recursive function calls. As the `mark_object` function introduced in Algorithm 1 is defined recursively, it has to be modified. One possibility is to explicitly manage the C-stack memory used by the recursive procedure. Another is to use a marking algorithm that uses a constant amount of space but has a worse computational complexity. Both solutions will be described in the following.

4.5.1 Depth-First Marking of Objects

The `mark_object` function in Algorithm 1 is performing a *depth-first* search [11] through the current reference graph $RG = (O_r, R)$. This algorithm runs asymptotically in time of

$$O(|O_r| + |R|)$$

and space of

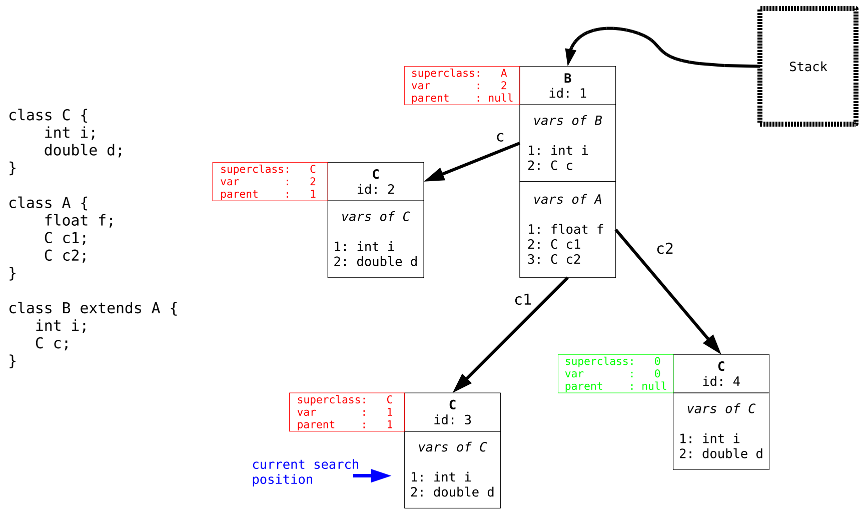
$$O(\text{length of longest simple path in } RG)$$

The required space is used to track the state of the search. In an implementation resembling Algorithm 1 it is implicitly used in the C-stack memory through the recursive function calls.

The worst case of space usage is when the longest simple path in *RG* contains all objects in O_r, i.e. the objects refer each other in a linked list as illustrated in Figure 9.

If a depth-first search algorithm for marking objects is used, the Java-VM has to deal with all possible reference graphs, including the worst case. This means that for each allocated object there has to be additional memory reserved to perform the marking procedure.

Each object in Java, besides holding the values of its instance variables resp. array fields, stores also some meta-information ,identifying for instance its class, used for look-up of virtual methods. It is stored in the so called *object header*. This object header is increased to hold the mark bit and the information necessarily to track the state of the search.



```
class C {
    int i;
    double d;
}

class A {
    float f;
    C c1;
    C c2;
}

class B extends A {
   int i;
   C c;
}
```

Figure 8: Depth first search for reachable objects

In Figure 8 an example of a search state is shown. The additional information to track the search is inside the colored boxes. A red box indicates that the object has been already marked, a green box means it hasn't. Currently the procedure is checking the first variable of object 3. There remains to check object 4 and its variables. The tracking information for each object consists of three parts:

1. `var`: the current variable
2. `super-class`: the current super-class of the current variable
3. `parent`: the previous object which is referring the object

When the procedure is finished checking the variables of object 3 it will return to its `parent`, object 1. There it will check the next variable, `var` 3 of `subclass` A. As this variable is a reference the procedure will descend into object 4, mark it, and initialize the tracking information (`subclass` C, `var` 1, `parent` 3). After object 4 has been processed the procedure returns to object 1 and can terminate because object 1 is directly reachable from a method stack.

An Algorithm describing this procedure of marking an object in more detail is given in Appendix B

4.5.2 Memory Overhead and Optimization Possibilities of Depth-First Marking

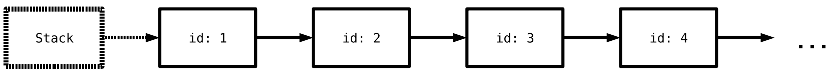


Figure 9: Worst case for depth first search

This above described procedure for marking objects introduces a considerable memory overhead to each object. It is in fact reserving the memory inside the objects for the worst case of a depth first search. This situation however is not typical.

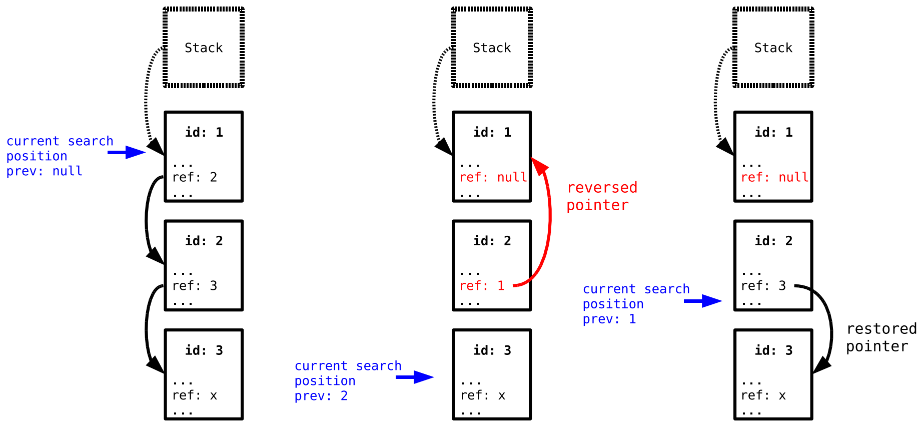


Figure 10: Pointer reversal

To reduce the memory overhead, it's possible to omit the explicit storage space for the `parent` object. This is a technique called *pointer reversal*[15] and is illustrated in Figure 10. While the procedure descends into an object through a variable of type `reference`, its possible to use this variable to store the reference to the parent of the object from which is being descended. The original value of the variable is restored when the procedure returns from the object to which it was descending.

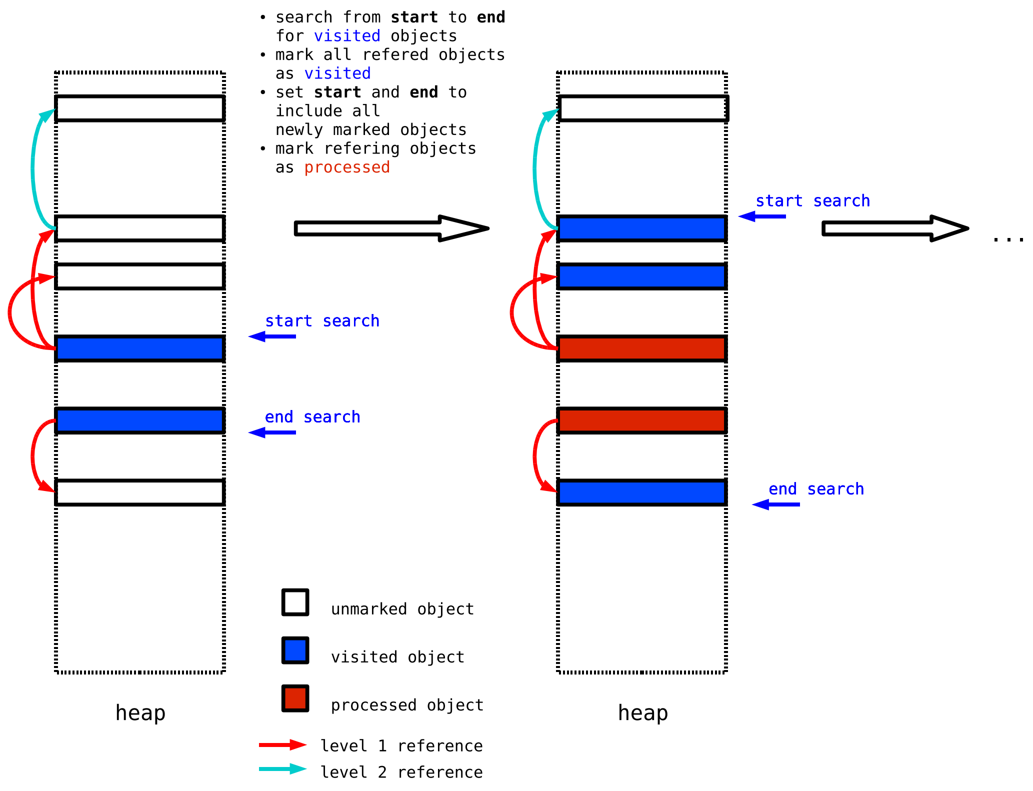

• search from **start** to **end** for visited objects
• mark all refered objects as visited
• set **start** and **end** to include all newly marked objects
• mark refering objects as processed
start search
end search
start search
end search
...
unmarked object
visited object
processed object
level 1 reference
level 2 reference
heap
heap

Figure 11: In-place marking principle

4.5.3 In-Place Marking of Objects

The memory overhead introduced by depth-first search can be avoided by using an alternative marking algorithm which runs in constant space. It is described in detail in Appendix C. Instead of doing a depth-first search to find all the objects transitively referenced by root set objects, each "depth-level" is checked separately by iterating through regions of the heap multiple times.

The principle is shown in Figure 11: at the beginning all the objects of the root set are marked as *visited*. During this process two pointers, `start` and `end` are updated to contain all objects that are visited. Then the procedure iterates through the region of `heap memory` defined by `start` and `end`. As it encounters the previously marked objects of the root set all the objects refered directly by them are now marked as visited. These are the *level 1* references in Figure 11. The *level 2* or further references are not followed, in contrast to depth-first search. Again the pointers `start` and `end` are updated to contain all the new visited objects. The previously visited objects are now marked as *processed* and are not considered further.

The process continues by iterating through the updated region until no more new objects are visited.

This algorithm only uses two kinds of marks in the *objects header*. No further dynamic memory, like an additional reference, is required. However it has a worse run-time complexity than the depth-first search as the heap, i.e. also the unreachable objects, has to be iterated for every reachable object in the worst case.

4.6 Integration of Garbage-Collector in Memory Management

When invoking the Garbage-Collector it has to search all reachable objects in the mark phase and then additionally iterate through all objects in the heap to check weather they are marked. This is a considerable effort which suggests to run the Garbage-Collector as little as possible. In the current implementation the Garbage-Collector is invoked automatically when no more memory is left for an allocation. As this could take quite some time where the Java program cannot be executed it is possible to invoke the Garbage-Collector through the library method `Runtim.gc()` from the program. This should be used by the programmer before calling code which should not be interrupted.

When Garbage-Collector finished freeing all unreachable objects the heap is left fragmented (i.e. there are free blocks inside the heap). This could lead to the situation that some new objects cannot be allocated because no free block is large enough to hold them. Then the Garbage-Collector would be invoked again even though there is free memory left inside heap memory. In order to try to postpone the next Garbage-Collector invocation the heap memory is compacted after freeing the unreachable objects as described in Section 4.3.3.

5 Stack Memory Management

As already described in Section 2.4 the other type of dynamically used memory of Java programs are the stacks of variable sized stack frames for each thread.

In a stack based instruction set like the Java bytecode instructions, stack operations a very frequently used. They should be executed with low computational overhead. Therefore the stack implementation should not split up the stack frames and specifically their operand stack as this would slow down those operations: popping a value from a separated operand stack for instance has to include additional checks to determine the actual segment of the opened stack which is currently used.

Method calls are also a very common operation in Java programs. Using numerous methods is considered good programming style. Therefore allocation should also be quick without much computational overhead.

Fast access and allocation of variable sized memory regions typically comes with a waste of memory. Two alternatives for stack management where implemented. The first uses linked fixed sized memory *chunks* that form the stack for a thread. It introduces a certain waste of memory. The second allocates chunks as special objects on the heap. They can have variable size. This will reduce the memory waste. The allocation of new chunks in this case however is only as fast as allocation of objects.

5.1 Fixed Sized Stack Chunks

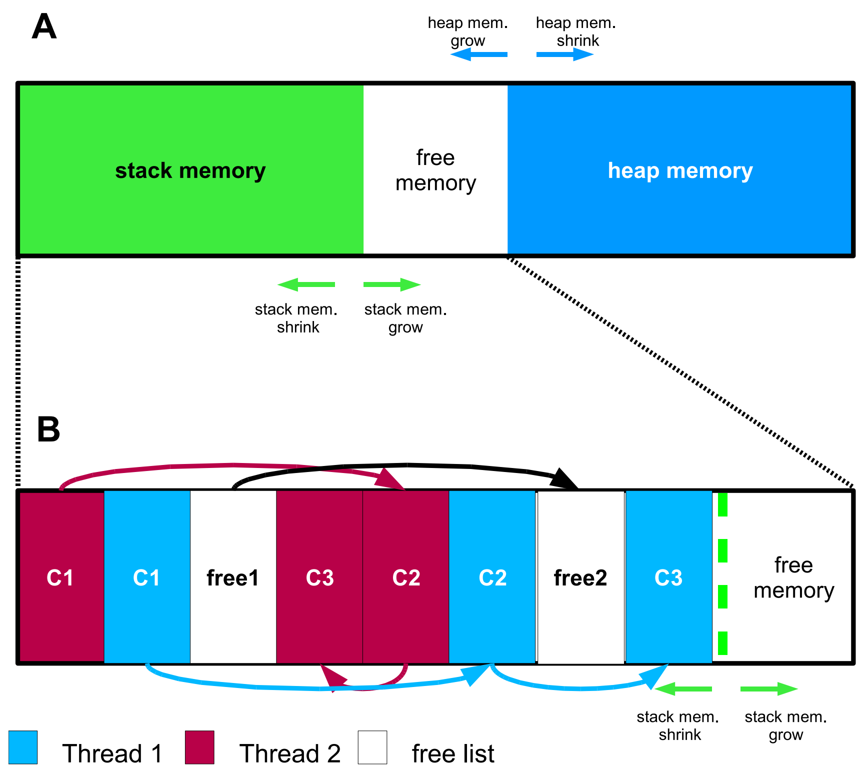


Figure 12: Layout of stack and heap when separated

With this approach a separate memory region, called *stack memory*, is used for the method stacks. This stack memory is placed at the opposite side of dynamic memory as the heap memory and is growing in opposite direction as illustrated in Figure 12A.

5.1.1 Allocation of Stack Chunks

When a thread becomes active (i.e. its `start`-method gets called) a memory region of fixed sized is assigned to the thread which should contain its stack. It is called a *stack chunk*. If the stack for this thread exceeds the amount of memory provided by the stack chunk then a new stack chunk has to be placed inside stack memory and linked to the previous stack chunk. Thus the stack of a thread is a linked list of stack chunks.

As the stack chunks are of equal size, they can be organized similarly to the entries of the reference table (which also are of fixed size) in Section 4.4.1:

- a free list for unused stack chunks is maintained
- new stack chunks are either taken from the free list (note that there is no need to search it as all chunks are of equal size) or stack memory is increased and the new chunk is placed at its end.
- when a stack chunk is freed it's either put into the free list or, if it's the top most chunks, stack memory is decreased.

This ensures rather fast allocation of stack chunks. Figure 12B shows a possible situation for the stack memory. There are two threads and each of their respective stacks occupies three stack chunks. Also there are two free stack chunks in the free list.

5.1.2 Collision with Heap Memory

If stack memory has to be increased but would reach into the heap memory then the Garbage-Collector is invoked. If after this there still isn't enough memory left to allocate the stack chunk, the Java-VM has to abort the method invocation and as a consequence the Java program.

When using a reference table the situation is slightly more complex: If stack memory has to be increased but would reach into the reference table then an attempt is made to move the reference table away from the stack memory in dynamic memory by the missing amount. If this is not possible because heap memory would then collide with the reference table even after Garbage-Collector invocation the method invocation is aborted.

As can be seen in Figure 12B the stack memory can be fragmented and it occupies a bigger region in the dynamic memory than necessary. The memory in the free list can always be used for stack chunks but not for objects.

Obviously the above described methods are opposed to the demand of a fast stack chunk allocation and should be considered a fail-safe mechanism.

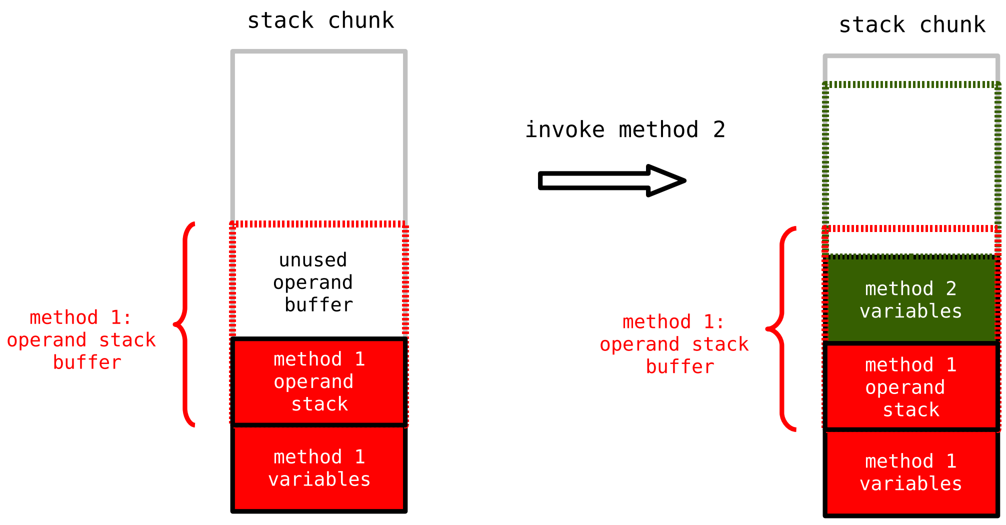


Figure 13: Push of stack frame inside a stack chunk

5.1.3 Push and Pop of Stack Frames

When a method is invoked during the execution of a thread than a memory region has to be provided to contain its stack frame. If there is enough space left in the stack chunk of the thread than the new frame is *pushed* on top of the current frame. The current frame will not be changed before the invoked method returns so the new frame can safely be placed on top of the *current* operand stack (Figure 13). This avoids wasting the unused part of the `operand-stack-buffer`. In the case that there is not enough space left in the stack chunk to hold the next frame a new stack chunk is allocated to hold the frame.

When a method returns and its stack frame is the only frame left in the stack chunk then this chunk is freed. A possible return value is put on top of the operand stack of the stack frame of the previous method (the one that has called the returning method).

5.1.4 Drawbacks

The major drawback of this approach is the waste of memory that occurs when a new stack frame doesn't fit inside of a chunk. In Algorithm 2, a procedure is described to calculate the memory wasted inside the stack chunks. It takes the size of a stack chunk "`size_chunk`" and a call chain "`call_chain`" as input. The maximum waste for a non recursive thread can then be calculated, for the chunk size s and all maximal call chains

$\{CC_1, \ldots, CC_m\}$ as

$$\text{waste}_{max} = max_i(\text{ waste}(s, CC_i)\ )$$

The maximum waste of the whole program cannot be decided at compile time as starting of new threads is run-time defendant.

Finding an optimal value for a given call graph is not trivial. One could assume that a large chunk size would reduce waste as waste increases with the number of chunks used to contain the whole call chain (see line 6-9 of Algorithm 2). However the waste that is added in line 13 is added for every thread and could cancel the benefit of large chunks for programs with numerous threads.

Algorithm 2 Calculating waste when using fixed sized stack chunks

```
 1  function waste( size_chunk, call_chain ):
 2      waste = 0
 3      chunk_count = 1
 4      current_size = 0
 5      for m in call_chain:
 6          if current_size + size( m ) > size_chunk:
 7              #use a new chunk as method cannot fit current chunk
 8              waste = waste + (chunk_size - current_size)
 9              current_size = 0
10          else:
11              #use old chunk
12              current_size = waste + size( m )
13      waste = waste + (chunk_size - current_size)
14      return waste
```

5.2 Variable Sized Stack Chunks

In order to avoid the memory related drawbacks of fixed stack chunks an alternative implementation of stack chunk allocation can be used. If the chunks are allowed to have variable size, the chunk can be truncated to the size that is currently occupied by stack frames. If a new chunk has to be allocated only the waste of the unused operand stack would be introduced for each chunk.

In Figure 14 an example is shown to illustrate the difference between fixed and variable sized stack chunks. In both cases the same four stack frames are on the stack of a thread. When using fixed sized chunks, the gray parts of the stack chunks are wasted as they are part of the chunk and cannot be used otherwise. The implementation using variable sized chunks

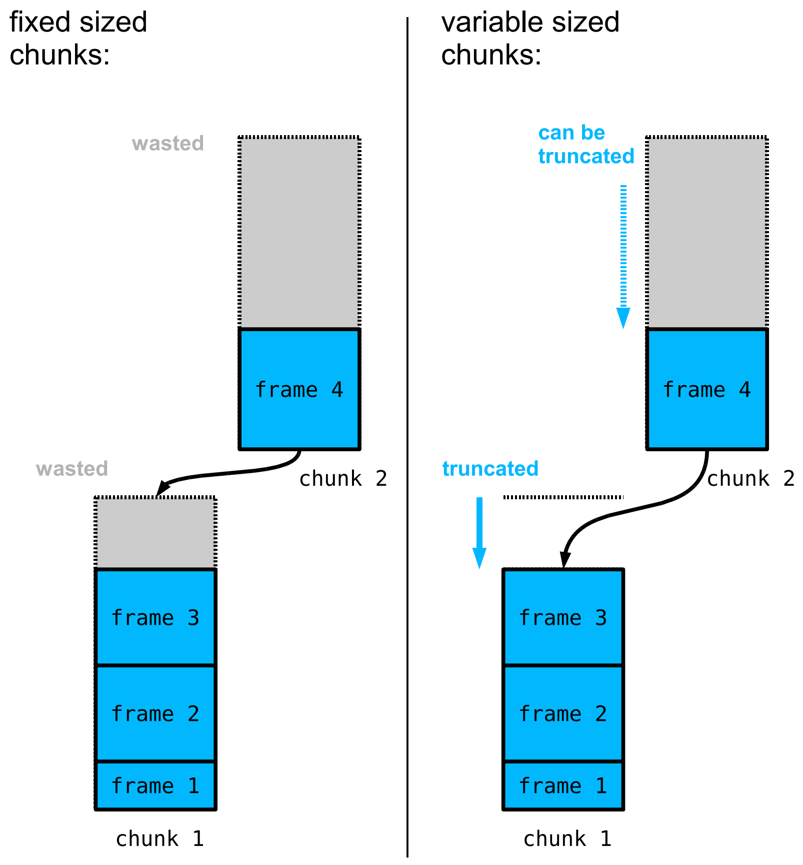


Figure 14: Fixed sized stack chunks compared to variable sized stack chunks

would truncate chunk 1 to avoid this waste. It is to be noted that also the top chunk could be truncated if the need occurs.

As management of variable sized chunks is in principle the same as the management of variable sized object the stack chunks are treated as special objects and allocated on the heap. This however would make allocation also as slow as allocating an object (searching the free list, interruption by Garbage-Collector) which is potentially in conflict with the design goal of having a fast method invocation.

6 Benchmarks and Results

The following section explains some test programs that were used to evaluate the performance and properties of the different memory management and Garbage-Collection approaches that were implemented during this thesis and discusses the results. At the time of writing nearly no real Java programs implementing typical sensor mote applications exist for *TakaTuka*. This is due to stability issues of the *TakaTuka* Java-VM, also arising from unstable memory management. This should change in the future as the foundation for stable memory management and Garbage-Collection is now implemented. The results given here should therefore be treated as preliminary. They probably cannot show the suitability of one or the other approach for real sensor mote applications.

First the to stack implementations, fixed size stack chunks and variable sized stack chunks, are compared. Then the two Garbage-Collection marking implementations, recursive and in-place marking, and the effect of compaction is examined.

All tests are performed by Java programs under the *TakaTuka* Java-VM either on a PC (Pentium 4, 3GHz, 1GB RAM running Linux) or a mica2 sensor mote. All speed related test where performed on the sensor mote as advanced hardware features of modern PCs and operating systems would distort the results.

For random number generation the pseudo random numbers from the `Random` class of the CLDC library were used.

6.1 Stack Implementation

As described in Section 5 there are two alternative implementations for managing the stack frames: Either fixed size stack chunks are allocated separately of the heap memory, abbreviated as *fixed-chunk* implementation in the following, or the stack chunks are allocated on the heap as objects and are truncated if there is potential memory waste (*variable-chunk* implementation).

Two types of tests were performed: A memory test performed on a PC that measures the memory usage of the stack and a speed test performed on a mica2 sensor mote.

6.1.1 Memory Usage

To get an impression of the memory usage of the stack management three different Java programs were run on the PC as there the internal state of the memory is easier accessible during the test. Because the memory management does not rely on library implementation, results obtained on a sensor mote should be comparable.

Two sizes of stack chunks were tested for each program and each stack implementation. In the case of the variable-chunk variant this is the amount allocated initially for each chunk, for the fixed-chunk variant this is the size of every chunk. All programs use the depth-first Garbage-Collector implementation. The specific sizes as well as the other memory related parameters are given in the description of the programs below.

The test measures the amount of memory that the stack would *at least* use for each method call or method return. Call and return of methods were chosen because these are the situations where the stack changes. "*At least*" means that for each method call/return the value is shown to which the stack *could be reduced if necessary*, i.e. if all of the remaining memory is needed by the program. The fixed-chunk variant cannot reduce its memory usage but for the variable-chunk it would mean that the most current chunk of each thread is truncated to the size of the stack frames it contains.

The programs used are

jvmTestCases: this is a Java program developed for testing the *TakaTuka* Java-VM functionality during development on the PC. It contains most instructions of the bytecode instruction set at least once. It features relatively large methods (stack frames with sizes exceeding 110 Bytes) and a lot of output. The program is single threaded.

The dynamic memory was set to 4096 Bytes. The stack sizes tested were 168 Bytes and 320 Bytes.

neighbourDiscoveryPc: this program was derived from a the Java program *neighbourDiscovery* which recognizes the surrounding sensor motes. It the first realistic application for *TakaTuka* on a mote. It was modified for the PC so that the messages normally received over the radio are now internally generated in regular intervals. There two threads running, one "receiving" the messages and the processing them.

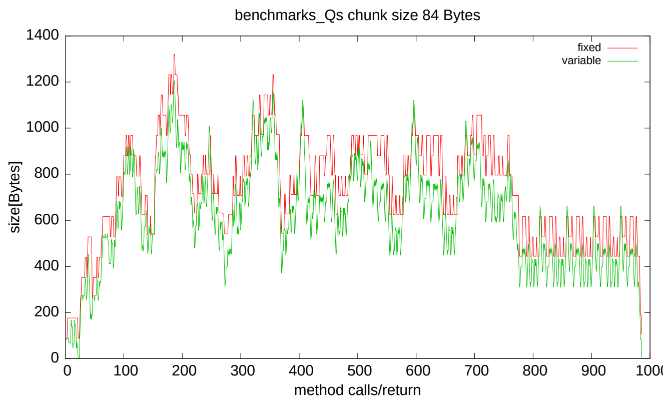
The dynamic memory was set to 2500 Bytes and the stack sizes tested were 84 bytes and 168 Bytes.

Qs: this is an artificial test program that creates arrays and sorts them. It does this with 3 threads running in parallel. Each creates an `int` array of a length randomly chosen from 1 to 20 and fills it with numbers from randomly chosen from 0 to 256. Then the array is sorted using a recursive form of the quick-sort algorithm.

The dynamic memory was set to 2500 Bytes and the stack sizes tested were 84 bytes and 168 Bytes.

Figure 15 shows the result of a test for two stack sizes with the Qs program. The results of the other programs can be seen in Appendix D. To summarize the behavior of the programs, Figure 16 shows *bars-and-whiskers* plots of these curves for all programs and conditions. In this representation the black bar denotes the median of the memory usage, the colored boxes indicate the lower and upper quartile and the lines mark minimum and maximum.

From these results it can be seen that the variable-chunk implementation occupies less memory than the fixed-chunk implementation. The difference



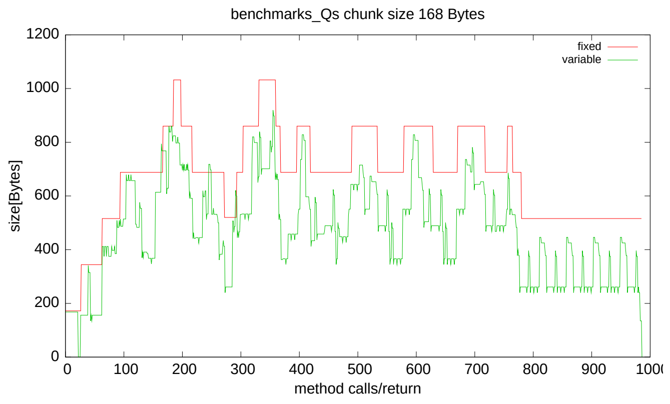


Figure 15: Memory usage over method call/return of Qs

gets considerably stronger when stack chunk size is increased. This is not surprising as big stack chunks reduce the memory waste in the lower stack chunks of the stacks for both implementations but increase the waste of the top stack chunks for the fixed size condition (see Section 5 and line 13 of Algorithm 2). For the chunk size of 84 Bytes the difference is moderate at least for program Qs.

6.1.2 Speed

A different set of programs were used to test the speed of the stack implementations. This is because jvmTestCases uses too much memory for the sensor mote and neighbourDiscovery is not adequate for speed tests as it contains arbitrary phases of inactivity. The following programs where used instead:

QsMote: a variation of the Qs program from the previous test. It differs in that the arrays are of length 1 to 10 and the test is repeated 30 times with varying seeds for the random numbers for each repetition.

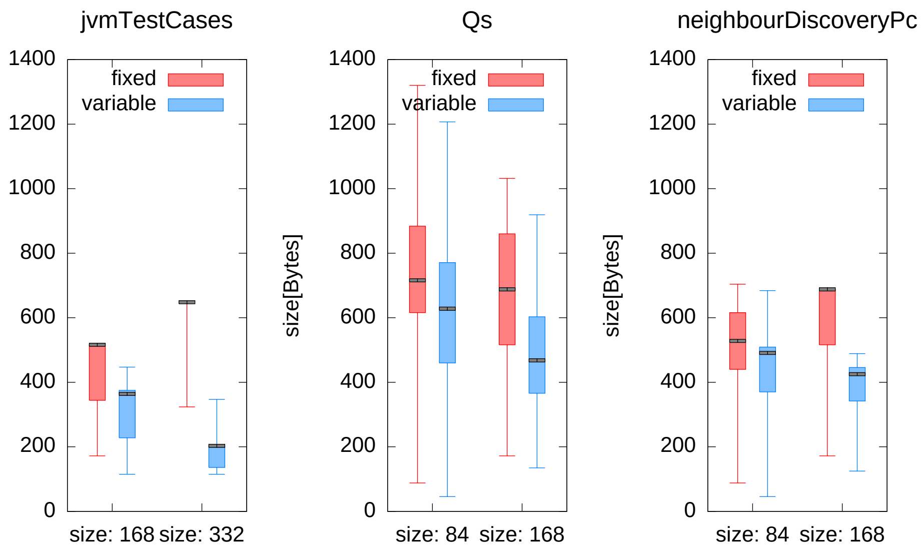


Figure 16: Bars-and-whiskers plots of stack memory usage

ChunkTest: this is another artificial program. It runs 3 threads which produce random numbers in the range of 0 to 16 which are then searched in an array of size 17 with a recursively implemented binary search. The main difference to QsMote is that no new objects are created during the process.

The results are shown in Figure 17. The speed gain achieved by the fixed-chunk implementation is very modest: only 7-10%. Probably the disadvantages of the variable-chunk approach, fragmentation of the heap and therefore slow allocation and higher number of Garbage-Collection invocations, do not kick in running the simple programs used for the test.

6.2 Garbage-Collector Implementation

The second set of tests concern the performance of Garbage-Collector implementation and the use of compaction. There are the depth-first implementation of the mark phase, denoted *mark-df* in the following, and the in-place implementation (*mark-in-place*). In addition there is the choice to use heap compaction with indirect references and a reference table or direct references.

The test performed was performed on the sensor mote with the dynamic memory size is set to 2500 Bytes. The variable-chunk stack implementation is used with an initial chunk size of 84 Bytes.

The test program, *GcTest*, creates small objects of a randomized size

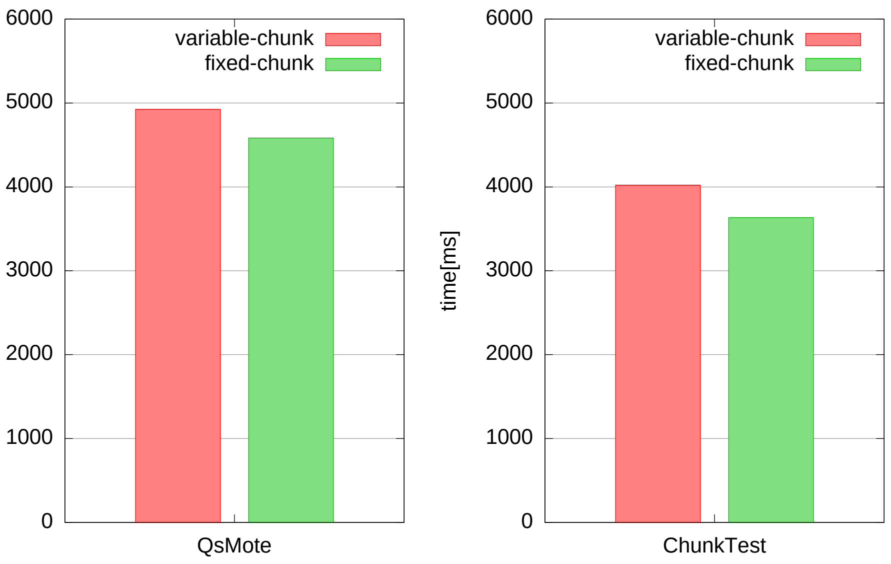


Figure 17: Speed measurements for the stack implementations

between 10 and 50 Bytes in a loop. Ten of those objects are always kept refered by an array in the root set. With a probability of 10% a newly created object is put into the array replacing the object which is already there. The test was performed with 500 and 2000 iterations and the number of Garbage-Collector invocations as well as the run-time were measured. The results are shown in Figure 18.

6.2.1 Results: Number of Garbage-Collections

In both tests (500 and 2000 iterations), the number of Garbage-Collections is greater when using direct references without compaction. Obviously the memory overhead introduced by the reference table does not outweigh the benefit of a compacted heap memory. In the case of 2000 iterations the difference is even more distinct which indicates an increase of fragmentation over time when not compacting the heap.

When considering the different Garbage-Collector implementations the results show that the memory overhead introduced by the depth-first implementation clearly leads to more Garbage-Collector invocations. As the objects in the test where relatively small, the overhead becomes more significant for more iterations if fragmentation is involved.

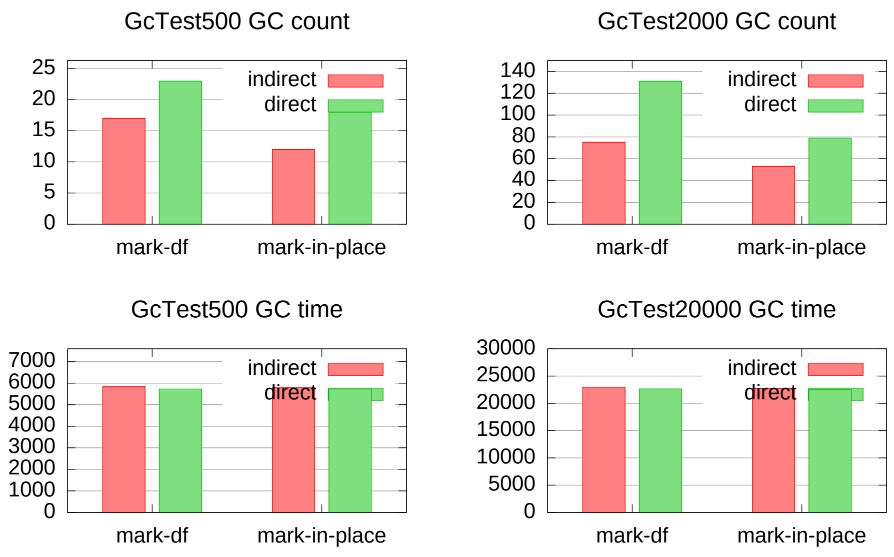


Figure 18: Results of the Garbage-Collector performance test

6.2.2 Results: Run-time

The run-time is not significantly affected by the choice of Garbage-Collector or whether direct or indirect references are used or not. The fact that there are only about 25 Garbage-Collector invocations and the program needs 6 seconds to terminate also indicates that the computational overhead of the Garbage-Collector is not critical for the run-time efficiency of the *TakaTuka* Java-VM. Probably the computational overhead of the memory management plays the dominant role for this test program.

7 Conclusion and Future Work

The main achievement of this bachelor's thesis is the implementation of functioning memory management and Garbage-Collection facilities for the *TakaTuka* Java-VM. From now on it is possible to develop realistic applications for sensor motes using object oriented programming style and design patterns. Prior to this thesis the use of dynamic memory, i.e. objects, in Java programs for *TakaTuka* had to be mostly avoided as the allocated memory couldn't be reclaimed.

The implementation exclusively uses C language constructs to access memory, i.e. no library functions or assembler code, and therefore should be portable to other sensor motes than the mica2.

The benchmark programs, that heavily use object allocation, could be run on the mica2 mote using 2500 Bytes of dynamic memory. However the performance results were not quite as expected and there is clearly the need to extend and/or modify the existing facilities and to test them with realistic WSN applications. Some propositions for future work are given in the following.

7.1 Improving Efficiency of Heap Memory Management

The results of Section 6.2.2 propose that the critical processes in the *TakaTuka* Java-VM aren't the Garbage-Collector but the allocation of objects and the execution of the bytecode. Before focusing on more advanced Garbage-Collector implementations for the Java-VM, these performance issues should be resolved.

A start for optimization could be the usage of a more elaborate type of data structure to search for free blocks. This would accelerate object allocation. An example would be a *segregated free list* [8] which allows direct access to blocks of different size. Also the ordering of free blocks could be varied.

7.2 Integration of Escape Analysis

The necessary extensions to the memory management to interpret the new `gc` bytecode instruction are straight forward. Especially for sensor motes the compile time identification of temporary objects could be very beneficial as it introduces very little computational overhead and the stop-the-world Garbage-Collection would only need to be run occasionally. Of course this depends on the ability to identify enough temporary objects at compile time which would have to be examined for typical sensor mote applications written in Java.

7.3 Variation of Garbage-Collector Integration

For real sensor mote applications it is probably suboptimal to run the Garbage-Collector only when the memory is exhausted. Its invocation should consider if the mote is in a phase of activity. For instance the Garbage-Collection could be triggered every time such a phase ends instead. Also compaction must not necessarily be coupled with Garbage-Collection.

A Appendix: Compaction Algortihm

This is the pseudo code for the compaction algorithm implemented. It compacts all used blocks after `first_free_block`.

```
#this function returns the next free block after memory_block
# and null if there is no more free block
function next_free_block( memory_block ):

#this function returns the next used block after memory_block
# and null if there is no more used block
function next_free_block( memory_block ):

# this function moves all blocks from src_start to one block before src_end
# to the position dest. It returns the next address after the moved blocks
function move( dest, src_start, src_end )


function compact( first_free_block ):

    move_start = next_used_block( first_free_block )
    while move_start != null:
        move_end = next_free_block( move_start )
        first_free_block = move( first_free_block, move_start, move_end )
        move_start = next_used_block( move_end )
```

B Appendix: Depth-First Marking Algorithm

This is the speudo code explaining the depth first marking algorithm. Each objects has an additional `search_info` field which contains

- the current superclass that is searched for fields `search_info.class`
- the fields not not searched yet for the current superclass `fields_left`
- the refernce to the parent objects `parent`

Objects belonging to the root set have parent **`null`**

```
function has_next_field( o ):
    while o.search_info.fields_left == 0:
        if o.search_info.class has a superclass:
            o.search_info.class = superclass( o.search_info.class )
            o.search_info.fields_left = field_count( o.search_info.class )
        else:
            #all classes have been searched.. no more fields left
            return false
    return true



function mark_object( ref ):

    while( true ):
        if ref == null:
            return
        o = lookup_object( ref )
        if o is numeric array:
            if o.marked == false:
                o.marked = true
            ref = o.search_info.parent
        elif o is reference array:
            #special treatment for refernce arrays:
            #set each entry as parent of the previous entry
            #the last entry gets ref as parent
            #...
        else:
            while true:
                if has_next_field( o ):
                    if next field is a reference:
                        next_ref = next_field( o )
                        next_o = lookup_object( next_ref )
                        next_o.search_info.parent = ref
                        ref = next_ref
                    else:
                        continue
                else:
                    ref = o.search_info.parent
```

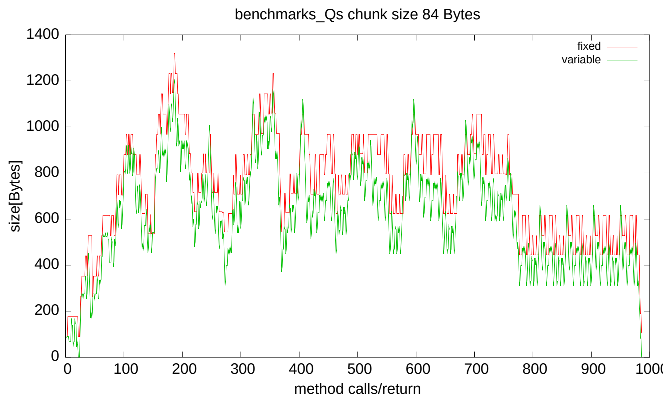
C Appendix: In-Place Marking Algorithm

This is the marking algorithm iterating through the heap instead of doing a depth first search. Each object has the additional fields `marked` and `visited`

```
#global variables indicating heap positions
start
end
objects_visited

function adjust( o ):
    if start > o:
        start = o
    elif end < 0:
        end = o

function visit( o ):
    if o.processed == true or o.visited == true:
        return
    o.visited = true
    objects_visited = true
    adjust( o )


function mark():
    for all objects o in the root set:
        visit( o )

    while objects_visited = true:
        for o between start,end:
            if o.visited = true:
                for c in objects refered by o:
                    visit( c )
                c.marked = true
```

D Appendix: Individual Stack Size Results

These are the stack size results on which the bars-and-whiskers plots are based. A stack size of zero is a measurement artefact, stemming from slightly different implementation of thread context switch in the variable-sized variant. They should be considered as outlayers and are not included in the bars-and-whiskers plots.



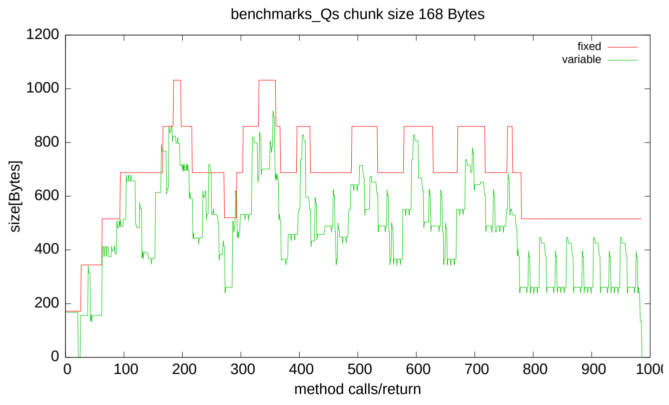
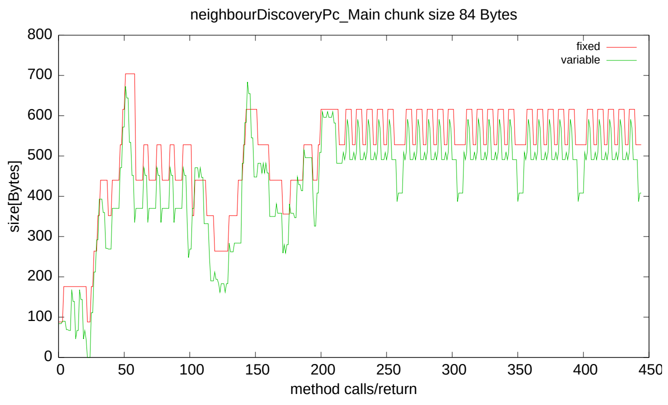


Figure 19: Qs



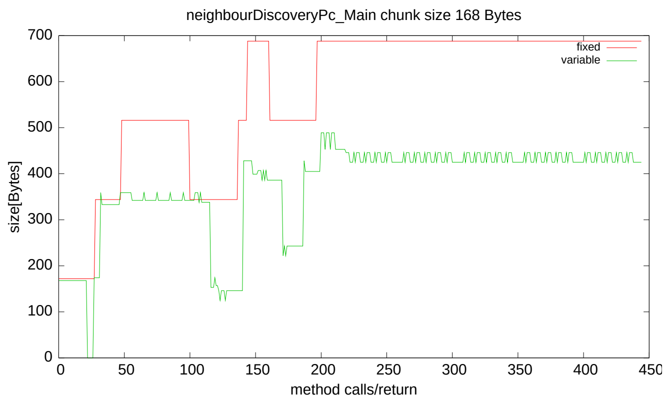


Figure 20: neighbourDiscoveryPc

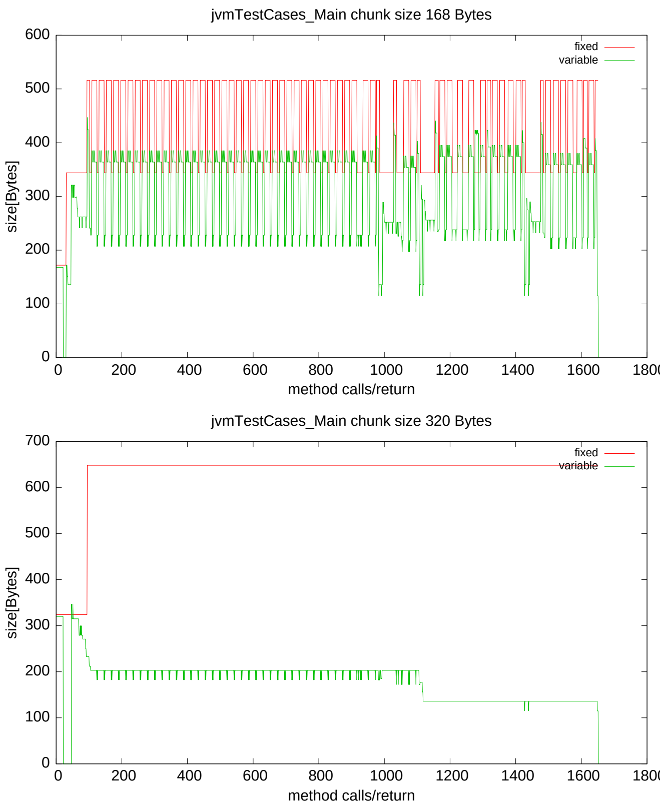


Figure 21: jvmTestCases

References

[1] Technical Advisor, Ken Arnold, Tim Lindholm, Frank Yellin, Frank Yellin, The Java Team, Mary Campione, Kathy Walrath, Patrick Chan, Rosanna Lee, Jonni Kanerva, James Gosling, James Gosling, James Gosling, James Gosling, Bill Joy, Bill Joy, Bill Joy, Guy Steele, Guy Steele, Gilad Bracha, and Gilad Bracha. The java language specification - second edition, 2000.

[2] Faisal Aslam, Christian Schindelhauer, Gidon Ernst, Damian Spyra, Jan Meyer, and Mohannad Zalloom. Introducing takatuka: a java virtualmachine for motes. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 399–400, New York, NY, USA, 2008. ACM.

[3] Avr-libc manual; memory areas and using malloc(). `http://www.nongnu.org/avr-libc/user-manual/malloc.html`.

[4] N. Brouwers, P. Corke, and K. Langendoen. Darjeeling, a java compatible virtual machine for microcontrollers. In *ACM/IFIP/USENIX 9th Int. Middleware Conference*, Leuven, Belgium, December 2008.

[5] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19, New York, NY, USA, 1999. ACM Press.

[6] 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.

[7] David Gay and Bjarne Steensgaard. Stack allocating objects in java (extended abstract).

[8] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *In International Symposium on Memory Management*, pages 26–36. ACM Press, 1997.

[9] Richard Jones and Rafael D. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, September 1996.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language / Brian W. Kernighan, Dennis M. Ritchie.* Prentice-Hall, Englewood Cliffs, N.J. :, 1978.

[11] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition).* Addison-Wesley Professional, April 1998.

[12] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition).* Prentice Hall PTR, April 1999.

[13] John L. Mccarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[14] Jan Meyer. Garbage collection for sensor motes. Bachelor's thesis, Albert-Ludwigs University of Freiburg, 2008.

[15] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

[16] Sun Microsystems, Inc. Connected limited device configuration (cldc); jsr 139. `http://java.sun.com/products/cldc/`.

[17] Sun Microsystems, Inc. The java hotspot performance engine architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

[18] Crossbow Technology. Mica2 wirelessm measurement system. Datenblatt Document Part Number: 6020-0042-08 Rev A, Crossbow Technology.

List of Figures