Abstract

Atmel is inventing a new microcontroller that is capable of running Java programs through an implementation of the Java Virtual Machine. Compared to industry standard PCs the microcontroller has limited processing power and main memory. When running interactive programs on this microcontroller it is important that the program interruption time is kept to a minimum.

In a Java Virtual machine the garbage collector is responsible for reclaiming unused main memory and making it available for the Java program again. This process creates a program interruption where the Java program is halted and the garbage collector is working.

At the project start the Atmel Virtual Machine was using the mark-sweep garbage collector. This garbage collector could produce a program interruption greater than one second and was not suitable for interactive programs.

The Memory-Constrained Copying algorithm is a new garbage collection algorithm that is incremental and therefore only collects a little bit of main memory at a time compared to the implemented mark-sweep garbage collector.

A theoretical comparison of the mark sweep algorithm and the Memory-Constrained Copying algorithm was performed. This comparison showed that the mark-sweep algorithm would have a much longer program interruption than the Memory-Constrained Copying algorithm. The two algorithms should in theory also produce equal throughput. The penalty for the short program interruption time in the Memory-Constrained Copying algorithm is its high algorithmic complexity.

After a few modifications to the Virtual Machine, the Memory-Constrained Copying algorithm was implemented and tested functionally. To test the program interruption and throughput of the garbage collection algorithms a set of benchmarks were chosen. The EDN Embedded Microprocessor Benchmark Consortium Java benchmark suite was selected as the most accurate benchmarks available.

The practical comparison of the two garbage collection algorithms showed that the theoretical comparison was correct. The mark-sweep algorithm produced in the worst case an interruption of 3 seconds, while the Memory-Constrained Copying algorithm's maximum program interruption was 44 milliseconds.

The results of the benchmarking confirms the results that the inventors of the Memory-Constrained Copying algorithm achieved in their test. Their test was not performed on a microcontroller, but on a standard desktop computer. This implementation has also confirmed that it is possible to implement the Memory-Constrained Copying algorithm in a microcontroller.

During the implementation of the Memory-Constrained Copying algorithm a hardware bug was found in the microcontroller. This bug was identified and reported so the hardware could be modified.

Contents

1 Introduction **1**

1.1 Project assignment 1

1.2 Outline 2

2 Background **3**

2.1 Java 3

2.1.1 Java editions 4

2.1.2 Running Java programs 7

2.2 Garbage collection and memory allocation 8

2.2.1 Example without garbage collection 9

2.2.2 Example with garbage collection 10

2.2.3 Garbage collection algorithms 10

2.3 The Memory-Constrained Copying algorithm 11

2.4 Microcontrollers 11

2.4.1 Java on a microcontroller 12

2.5 Atmel's 32-bit microcontroller 13

2.5.1 Atmel's Java Virtual Machine 14

2.6 Background summary 15

3 A theoretical comparison between the mark-sweep algorithm and the MC^2 algorithm **17**

3.1 A note about pseudo-code 17

3.2 The mark-sweep algorithm 18

3.2.1 Heap layout 18

3.2.2 Memory allocation 19

3.2.3 Garbage collection 20

3.3 The Memory-Constrained Copying algorithm 22

3.3.1 Heap layout 22

3.3.2 Memory allocation 23

3.3.3 Garbage collection 23
3.4 Comparison 25
3.4.1 Memory utilization 25
3.4.2 Program interruption 26
3.4.3 Throughput 26
3.4.4 Algorithmic complexity 27
3.5 Comparison summary 27

4 Preparations before implementing the MC^2 algorithm 35
4.1 Java memory interface 35
4.1.1 Objects and arrays 35
4.1.2 Object handles 35
4.1.3 Special Java instructions 39
4.2 Modifications in the AVM 45
4.2.1 Make the array and object structure similar 45
4.2.2 One common function for allocating objects and handles . 45
4.2.3 Fixing the mark-sweep implementation 47
4.3 Preparation summary 48

5 Designing and implementing the Memory-Constrained Copying garbage collector 49
5.1 Heap layout 49
5.1.1 Data structure to manage the heap windows 49
5.1.2 Managing object handles 50
5.1.3 Handling objects larger than one window 51
5.1.4 Immortal objects 52
5.1.5 Heap management 52
5.1.6 Initial number and size of windows 54
5.1.7 Heap windows summary 54
5.2 Write barrier trap 54
5.3 Locating object pointers 55
5.4 Object allocation 56
5.4.1 Allocating memory from the native library 56
5.4.2 Locating nursery handles 56
5.5 Nursery collection 57
5.6 Old generation marking 58
5.6.1 Mark stack 59
5.7 Old generation copying 60

5.8 User triggered garbage collection 60
5.9 Remembered sets, sequential store buffers and card tables 61
5.10 Source code 61
5.11 Implementation summary 61

6 Testing the garbage collector implementations 63
6.1 Test setup 63
6.1.1 Hardware-software co-simulation 63
6.1.2 Software simulator 64
6.1.3 Hardware simulator 66
6.2 Testing procedures 67
6.2.1 Test programs 67
6.2.2 Functional testing 68
6.2.3 Test method 70
6.3 Testing results 71
6.4 Test summary 71

7 Choosing a garbage collection benchmark 73
7.1 Benchmarking and benchmarks 73
7.1.1 Benchmark categories 74
7.1.2 Benchmark suites 75
7.2 What to measure 75
7.2.1 Program interruption 75
7.2.2 Throughput 76
7.2.3 Factors that influence the measuring 76
7.3 Benchmark candidates 76
7.3.1 Write a self-composed benchmark 77
7.3.2 The EEMBC Java benchmark suite 77
7.3.3 The CaffeineMark benchmarks 78
7.4 Choosing a benchmark 79
7.5 How to measure program interruption and throughput 80
7.5.1 Program interruption 80
7.5.2 Throughput 80
7.6 Choosing benchmark summary 81

8 A comparison between the mark-sweep and the MC^2 implementations **83**
8.1 Test system . . . 83
8.2 Test parameters . . . 83
8.2.1 Heap size . . . 84
8.2.2 Memory-Constrained Copying heap window size . . . 84
8.3 Test cases . . . 84
8.4 Test results . . . 85
8.4.1 Program interruption time . . . 85
8.4.2 Throughput . . . 87
8.5 Benchmark result summary . . . 88

9 Conclusion **91**
9.1 Main contributions . . . 91
9.1.1 The MC^2 algorithm has been implemented . . . 91
9.1.2 The MC^2 article's results have been confirmed . . . 92
9.1.3 The mark-sweep implementation is working again . . . 92
9.1.4 The AVM has been improved . . . 92
9.1.5 Hardware bug found . . . 93
9.2 Project Experiences . . . 93
9.2.1 Implementation challenges . . . 93
9.2.2 Status meetings . . . 95
9.2.3 The project schedule . . . 95
9.3 Future work . . . 96
9.3.1 Optimize the MC^2 implementation . . . 96
9.3.2 Make the garbage collectors throw an OutOfMemoryException . . . 97
9.3.3 Test the MC^2 implementation against the remaining benchmarks . . . 97
9.4 Conclusion summary . . . 97

Bibliography **99**

Appendixes **101**

A Garbage collector code **103**

B Microcontroller white paper **105**

List of Figures

2.1 Difference in API between Java editions. 4
2.2 Relationship between profiles and configurations in J2ME. 5
2.3 Compiling and running a Java program. 7
2.4 Memory management without garbage collection. 9
2.5 Memory management with garbage collection. 10
2.6 Example block diagram of a microcontroller. 12
2.7 Java traps. 14
2.8 Compiling and running a Java program for the AVM. 15
2.9 The relationship between the AVM and the JEM. 16

3.1 Heap organization with the mark-sweep algorithm. 18
3.2 Fragmentation example. 20
3.3 Pointer reversal example. 21
3.4 Heap organization with the MC^2 algorithm. 23
3.5 MC^2 garbage collection example. 24
3.6 Program interruption comparison. 33

4.1 Object and array structure. 36
4.2 Object handles. 37
4.3 Object handle with array, immortal and mark bit. 37
4.4 Code for the `putfield` instruction. 38
4.5 `putfield` example without handles. 39
4.6 `putfield` example with handles. 40
4.7 Error during incremental marking. 41
4.8 code for the MC^2 write barrier. 41
4.9 `aputfield_quick` example with write barrier. 43
4.10 Code for the `aputfield_quick` instruction. 44
4.11 The new object and array structure. 46

5.1 Objects and handles on the heap without garbage collection. . . 50
5.2 Handle free list example. 51
5.3 Heap defragmentation example. 53
5.4 Object handle with array, immortal, mark and nursery mark bit. 55
5.5 Heap example with nursery handle pointers. 57

6.1 Software simulator screen shot. 65
6.2 The hardware test setup. 66
6.3 Flow diagram of the garbage collector test method. 70

8.1 Chess benchmark test results with 500 kB heap. 89
8.2 Throughput result from the chess benchmark. 90

9.1 Project schedule. 95

A.1 CD contents. 103

List of Tables

3.1 Summary of the comparison between the mark-sweep and the MC^2 algorithm. 32

4.1 The object header items. 36
4.2 The array header items. 37
4.3 The object handle status bits. 38
4.4 Bytecodes that stores object pointers. 42
4.5 The new object and array header items. 46

5.1 The heap window data structure. 50
5.2 Summary of window types. 54

6.1 The relationship between the simulators and the hardware-software co-simulation levels. 64

7.1 Program interruption limits. 76
7.2 Benchmark alternatives. 77

8.1 Test cases for garbage collection benchmarking. 85
8.2 Program interruption results. 86
8.3 Throughput results. 87

List of Algorithms

3.1 Mark-sweep: Pseudo-code for the allocate procedure. 19
3.2 Mark-sweep: Pseudo-code for the gc procedure. 19
3.3 Mark-sweep: Pseudo-code for the mark procedure. 28
3.4 Mark-sweep: Pseudo-code for the sweep procedure. 28
3.5 Mark-sweep: Pseudo-code for the cleanup procedure. 29
3.6 MC^2: Pseudo-code for the writeBarrierTriggered function. 29
3.7 MC^2: Pseudo-code for the allocate function. 29
3.8 MC^2: Pseudo-code for the nurseryCollect function. 30
3.9 MC^2: Pseudo-code for the oldGenerationMark function. 31
3.10 MC^2: Pseudo-code for the oldGenerationCopy function. 31
3.11 MC^2: Pseudo-code for the gc function. 32

Chapter 1

Introduction

This report begins with the project assignment and is followed by an outline of the following chapters.

1.1 Project assignment

The project assignment can be found below and is followed in this report without any additions or corrections.

> "Java has, in contrast to for example C++, automatic allocation and deallocation of memory. To be able to reuse memory, objects with no references have to be collected and reassigned to free memory. This process is known as garbage collection. Garbage collection is closely tied with memory allocation, since garbage collection often is part of or cooperating with memory allocation.
>
> Atmel's new microcontroller is able to run Java programs by an implementation of the Java Virtual Machine (JVM). This JVM implementation collects memory with the mark-sweep garbage collector. This garbage collection algorithm needs to stop the execution of Java programs for a relatively long time while collecting garbage. For a microcontroller designed to run interactive Java programs, e.g. games, this is not preferable.
>
> Memory-Constrained Copying (MC^2) is a garbage collection algorithm designed to be used in a microcontroller with a small memory that runs interactive programs. The algorithm is incremental and collects just a little bit of the Java memory at a time. This leads to shorter program interruptions.
>
> The project contains the following tasks:
>
> - Compare the mark-sweep and MC^2 garbage collection algorithms.
> - Implement the MC^2 algorithm in the Atmel Java Virtual Machine.

- Test the implementation.
- Choose a set of relevant applications and benchmarks.
- Compare the program interruption and throughput between the MC^2 and the mark-sweep implementations, using the chosen applications and benchmarks."

1.2 Outline

This report is partitioned in two parts. The first part consists of the report itself, while the second part includes the source code for the MC^2 garbage collector and the white paper for the microcontroller. The second part is not available for the general public.

This report closely follows the sequence of tasks from the project assignment and the chronological sequence of the work. The chronological sequence can be seen in the project schedule in figure 9.1 on page 95. The only exception to this is the mark-sweep implementation changes. These are included in chapter 4, although they were completed after the benchmarks were chosen.

After this introduction this report continues with a background chapter, chapter 2. This chapter will provide information about key concepts from the project assignment.

In chapter 3 the mark-sweep and MC^2 algorithm are described and compared on a theoretical basis. This chapter also includes a pseudo-code for the algorithms.

A description of the microcontrollers memory system and the modifications of the Atmel Virtual machine are included in chapter 4. The modifications of the mark-sweep implementation is also part of this chapter.

The design choices and implemented solutions when implementing the MC^2 algorithm are presented in chapter 5.

Chapter 6 describes the test system and the functional test method used to test that the MC^2 implementation was correct. A description of the test programs and test method are also included in this chapter.

In chapter 7 a set of benchmarks for comparing the program interruption and throughput of the garbage collection implementations are chosen. This chapter begins with a selection of candidate benchmarks and is followed by an evaluation of these before a suitable candidate is chosen.

The results of the benchmarking are found in chapter 8. This chapter includes a set of test cases and the program interruption time and throughput results of these tests.

The last chapter in this part of the report, chapter 9, contains the conclusion. The results, a project experiences and possible future work can be found here.

The second part begins with a description of the files included on the source code CD, in appendix A. The CD is located in a folder on the last page of this part.

The last appendix, appendix B, contains a white paper for the new Atmel microcontroller.

Chapter 2

Background

This background chapter begins with an introduction to some key concepts from the project assignment, like Java, garbage collection and microcontrollers. A short introduction to the Memory-Constrained Copying algorithm is also included.

The introduction was originally written in the project leading to this diploma [Amu04], except the Memory-Constrained Copying section. The introduction found here is a revised edition of the one found in the project report.

2.1 Java

The Memory-Constrained Copying garbage collection algorithm will be implemented in a Java Virtual Machine that implements a specific Java edition. The Java Virtual Machine runs on a microcontroller that is able to run Java code. For readers unfamiliar with the Java programming language, Java editions and the Java Virtual Machine, a short introduction is included here.

This information was gathered from Russel Winder and Graham Roberts book "Developing Java Software, 2nd edition" [WR00], except the Java editions section, section 2.1.1. The source of this subsection is "J2ME In A Nutshell, A Desktop Quick Reference", by Kim Topley [Top02].

The Java programming language was developed by James Gosling and other engineers from Sun Microsystems. Java was originally developed to replace C++ and had these key features:

Object orientation This is a programming language design that aims to make the programmer represent real-world objects as objects in the programming language.

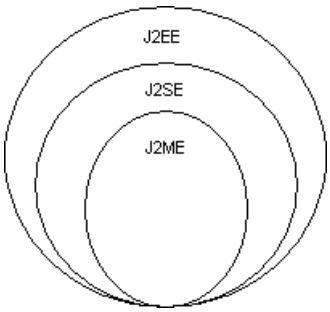
Platform independence Java programs can be executed on almost every platform. This is possible by compiling the Java code to an intermediate Java bytecode. This Java bytecode is then interpreted in a special program, called the Java Virtual Machine, on the target platform.

Rich library with networking functions The Java Application Programming Interface (API) comes with many library classes, supporting features like: networking, input/output, graphical user interface libraries and much more. The Java API has grown from around 100 classes in the earliest version (Java version 1.0) to almost 3000 classes in Java 2 version 5, the newest version.

Execute remote code securely Java also includes a framework for executing code outside the machine the main program runs on. This enables the creation of a distributed system. The creators have also taken measures to make the execution secure.

In addition to these key features, the Java language is also a garbage collected language. This means that garbage collection (See section 2.2) is a specified feature of the language.

2.1.1 Java editions



This figure is based on a figure from [Top02].

Figure 2.1: Difference in API between Java editions.

Java is distributed in three editions, targeting many different devices from mobile phones to big enterprise servers. The main difference between these editions is in the Java API (See figure 2.1). The three editions are:

Java 2 Enterprise Edition (J2EE)

The enterprise edition is targeting business applications and has a very rich API library. This library consists of the standard J2SE API, SQL and transaction support, XML support and servlet support to name a few features.

Java 2 Standard Edition (J2SE)

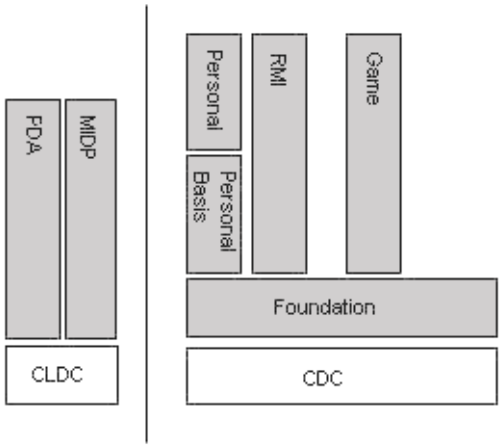
This Java edition is targeting normal workstation environments and is the main Java edition.

Java 2 Micro Edition (J2ME)

The J2ME is a stripped down version of the J2SE. It targets mobile phones, PDAs and other products with a small amount of memory and limited processing power.

The J2ME market is wide and rich. It was therefore necessary to create two subsets of the J2ME. A specification of such a subset is called a configuration. This configuration also includes a specification of what capabilities the processor running Java has. Today there exists two J2ME configurations: The Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC).

In addition to the configurations there exists several profiles. The profiles extends the functionality of the configurations to make the J2ME usable in different settings. This arrangement and the existing profiles are shown in figure 2.2.



Profiles are colored grey and configurations are white. This figure is based on a figure from [Top02].

Figure 2.2: Relationship between profiles and configurations in J2ME.

Connected Limited Device Configuration

A typical CLDC device has minimum 128 kB of ROM, flash memory or RAM to store the Virtual Machine and Java Program code in. It also has at least 32 kB of RAM to be used by the Virtual Machine.

In the CLDC specification the following features have been removed from the standard Java API (J2SE) and the Virtual Machine:

Reflection Reflection is a tool that allow programs to determine the class of an object, get the methods, fields and constructors of a class and allow the program to run methods in classes that are not known until runtime.

Weak references Weak references offers the user programs to interact with the garbage collector in a limited way. An object pointed to by *only* a weak reference is always collected when running garbage collection.

Java Native Interface The Java Native Interface (JNI) offers the user to write code that interacts with the target platform running the Java Virtual Machine. The user writes code (typically in C) and can execute this by invoking the function as a normal Java method. Native functions still exists in the library classes, but the user's interface for native functions are removed.

Deamon threads and thread groups Deamon threads are automatically terminated if all other non-deamon threads are terminated in the JVM. Thread groups offers a way to group threads and starting and stopping these groups.

Object Finalization When the garbage collector collects an object it is supposed to run the `finalize()` method on all objects. This could in the worst case lead to that the object resurrects itself by calling a method that stores a pointer to this object.

Error and exception handling The normal JVM contains many error and exception classes. These classes are not essential because the JVM never can recover from an error situation. To save memory these classes have been removed.

Floating point instructions These instructions are removed from the CLDC specification of the Java Virtual Machine.

Currently there exists only two CLDC profiles:

Mobile Information Device Profile (MIDP) The MIDP is a CLDC profile specially targeting mobile phones and pagers, but also smaller PDAs. It extends the CLDC API with a special MIDlet environment, user interface, networking and persistent storage. An application that runs on a MIDP device is called a MIDlet.

PDA profile The PDA profile is similar to MIDP, but is targeting PDAs with larger screens and more main memory. The PDA profile makes it possible to build more sophisticated user interfaces than the MIDP.

Connected Device Configuration

The CDC profile is targeted at devices that has a minimum of 2 MB of memory and a 32-bit processor. The device is also often directly connected to the Internet or a local intranet.

A CDC Virtual Machine must implement all the features of the Java 2 Virtual Machine specification [LY99], but have some small differences in the API.

CDC profiles:

Foundation profile This profile is the foundation for most of the other CDC profiles. This profile extends the J2ME CDC API to contain almost all classes from the J2SE API, except the GUI (Graphical User Interface) and RMI (described below) classes.

RMI profile RMI stands for Remote Method Invocation. This profile adds a subset of the J2SE RMI package to the CDC. The RMI package allows a RMI client to invoke Java methods on an RMI server. Only the client RMI functions are included, because a CDC device often is used in the role of a RMI client.

Personal profile The personal profile provides the CDC with a graphical user interface. This profile is under development.

Game profile The game profile is intended to support gaming software on a CDC device. This profile is under development.

2.1.2 Running Java programs

To run programs written in Java code, the program first have to be compiled into a Java classfile containing Java bytecodes. The classfile then has to be executed in a Java Virtual Machine implementation on the target platform. This process is shown in figure 2.3.

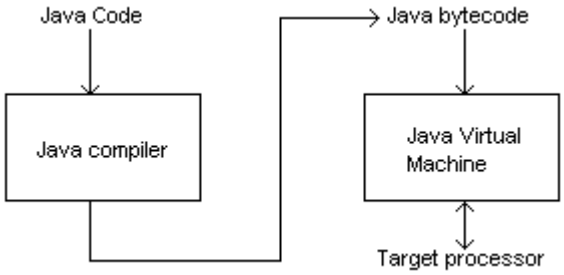


Figure 2.3: Compiling and running a Java program.

Java Virtual Machine

Java programs are, when compiled, transformed into class files containing Java bytecodes. To be able to run these Java bytecodes on the target platform, the platform must have an implementation of the Java Virtual Machine. The specification of the Java Virtual Machine can be found in the book "The Java Virtual Machine Specification, second edition" by Lindholm and Yellin [LY99].

The Java Virtual Machine interprets the Java bytecodes and transforms these into machine instructions that the target platform can understand and execute. Prior to the execution of the program the Java Virtual Machine has to do the following:

- Load the class files and API library.

- Verify the loaded classes.
- Initialize the system library classes.

While running Java bytecodes, the Virtual Machine has some responsibilities: It must perform method invocation, run native methods, perform memory allocation and run the garbage collector when necessary.

The Java bytecode instruction set does not provide input/output instructions or memory mapped input/output of any kind. Therefore to be able to communicate with its surroundings the Java Virtual Machine includes something called Java native methods.

These methods can be called like any other Java method, but they consist of code that is native to the platform the Java Virtual Machine is executed on (often compiled C code). These methods provide the JVM with the ability to produce textual output or read input from the keyboard, for instance.

2.2 Garbage collection and memory allocation

To understand the different types of garbage collection algorithms one have to have a understanding of what garbage collection is. A short introduction to garbage collection is therefore included in this section. This information is based on [mem05].

Some programming languages, like C, have explicit memory management. In an explicit memory management system, it is the programmer's responsibility to allocate and deallocate memory as needed. Many programming errors stem from memory allocation/deallocation errors and pointer management. These errors can be tricky to locate, if your program is of some complexity. Therefore implicit memory allocation (Garbage collection) was introduced to assist the programmer. Garbage collection was invented by John McCarty as a part of the first Lisp system.

In a programming language that supports garbage collection, memory management becomes an automated task, carried out by the runtime system. The programmer still has to allocate memory, but deallocation is now carried out by the system.

Deallocation of memory is carried out either when the memory is full during an allocation or at regular intervals. The garbage collector have to scan through all the references in the system, looking for blocks in the memory that is unreachable. An unreachable block is an object that no references point to. If a block is unreachable, then it is of no use to the user program and can be reassigned to the pool of free memory. A block that is reachable is called a live block.

Sometimes it can be difficult to separate a reference from an integer, for instance. This is especially the case when the runtime system does not keep information about the type of the variable. If this is the case the garbage collector needs to use a conservative estimate to establish the amount of free memory. This is called a conservative collector. The opposite, an exact collector, can calculate the memory exactly.

2.2.1 Example without garbage collection

This section includes an example which illustrates the problems that can occur without garbage collection. The example program allocates memory for an array containing ten integers. This array will only be used inside the procedure and will be deallocated explicitly at exit. Example code in C is shown in figure 2.4.

```
1.  void main(){
2.    int *array;
3.    array = (int *)malloc(sizeof(int)*10);
...
10.   array[0] = 10;
11.   array[1] = 9;
...
20.   printf("%i\n" , array[0]);
21.   printf("%i\n" , array[1]);
...
30.   free(array);
31.   return;
32. }
```

Figure 2.4: Memory management without garbage collection.

In this example an array, capable of holding 10 integers, is allocated from the heap (Line 3). Later the array is initialized with values, from line number 10. Then these values are displayed with the `printf()` function (Line 20). Before returning from the function, the memory that kept the array is deallocated using `free()` in line 30.

If the programmer does not deallocate the memory before exiting (by omitting line 30), the program never reclaims the array. This is often called a memory leak. If several memory leaks exists in the program, it may run out of memory after a while.

Another problem occurs if the programmer have changed the pointer by accident (For example by replacing line 20 and 21 with `printf("%i\n" , *(array++));`) and then deallocated it, believing that it was the original pointer that was deallocated. This often damages the internal structures the runtime system uses to keep track of free and used memory. This will confuse the allocator when a new memory block is allocated and the program would eventually crash.

If a chunk of memory is deallocated prematurely (by calling `free()` on line 19 instead of line 30, for instance), it creates a dangling pointer (A pointer that points to unreliable data.). When the program tries to access the object that the dangling pointer points to, after the object was deallocated, the object may still be intact. But at a later stage this part of the memory could have been overwritten. If the program then accesses the object, it will read data that are incorrect and probably fail.

In a more complex program these errors can be hard to find, because the consequences can occur at a much later stage (or not at all) and not directly

after executing the faulty line of code.

2.2.2 Example with garbage collection

This is the same example as the previous, but the code is written in Java, a language that supports garbage collection (See code in figure 2.5). Although the code in this example looks somewhat different than in the previous example, all the lines in this example are semantically equivalent to the corresponding lines in the previous example.

```
1.  void main(String[] args){
2.    int[] array;
3.    array = new int[10];
...
10.   array[0] = 10;
11.   array[1] = 9;
...
20.   System.out.println(array[0]);
21.   System.out.println(array[1]);
...
31.   return;
32. }
```

Figure 2.5: Memory management with garbage collection.

Notice that in this example line number 30 is missing. In the previous example this line deallocated the array of integers.

After this method is executed the garbage collector notices that the array is inaccessible. It is inaccessible because no pointers exists in the system that is a reference to the array. The only reference existed inside the scope of the method and were not returned or stored in any way. The garbage collector can then reclaim this memory and make it available for reuse when invoked. Notice that the programmer does not need to invoke the garbage collector explicitly.

Memory leaks are avoided because the programmer never have to free the objects explicitly. This task is the runtime system's responsibility and it will never reclaim an object if there still exist a pointer to it. Dangling pointers are also impossible by the same reason.

2.2.3 Garbage collection algorithms

There are two main types of garbage collection algorithms, tracing algorithms and reference counting algorithms:

Tracing algorithms Tracing algorithms focus on determining which objects are reachable and then reclaim all the other objects. This is the most common type of garbage collector implemented.

Reference counting algorithms These algorithms focus on determine when objects become unreachable and collects these when it happens. Reference counting keeps track of live objects using a counter that counts references to this object. If the count reaches zero, the object is unreachable and is reclaimed by the garbage collector.

Both the mark-sweep and Memory-Constrained Copying algorithms are tracing algorithms.

2.3 The Memory-Constrained Copying algorithm

The increasing popularity of Java enable handheld devices, like mobile phones and PDAs, has created a need for a high performance garbage collector for devices with a limited amount of processing power and main memory. The most popular algorithms for such devices are mark-sweep and mark-compact algorithms [SMB04]. These algorithms suffer from high program interruption time.

The MC^2 algorithm is a new garbage collection algorithm designed specifically for embedded devices that runs interactive programs. It was invented by Sachindran, Moss and Berger at the university of Massachusetts. The article was published in the beginning of 2004 [SMB04].

The only known implementation of this algorithm is the inventor's test implementation in the Jikes Research Virtual Machine (implementing the J2SE API) developed by IBM. This runs on a modern desktop computer with a Pentium 4 processor and 512 MB RAM.

The MC^2 algorithm has not been tested yet in a real microcontroller with a Java Virtual Machine implementing the J2ME API. This implementation will therefore be the first run a real embedded system and will make it possible to confirm the results presented in the article [SMB04].

The Memory-Constrained Copying algorithm details are presented in section 3.3.

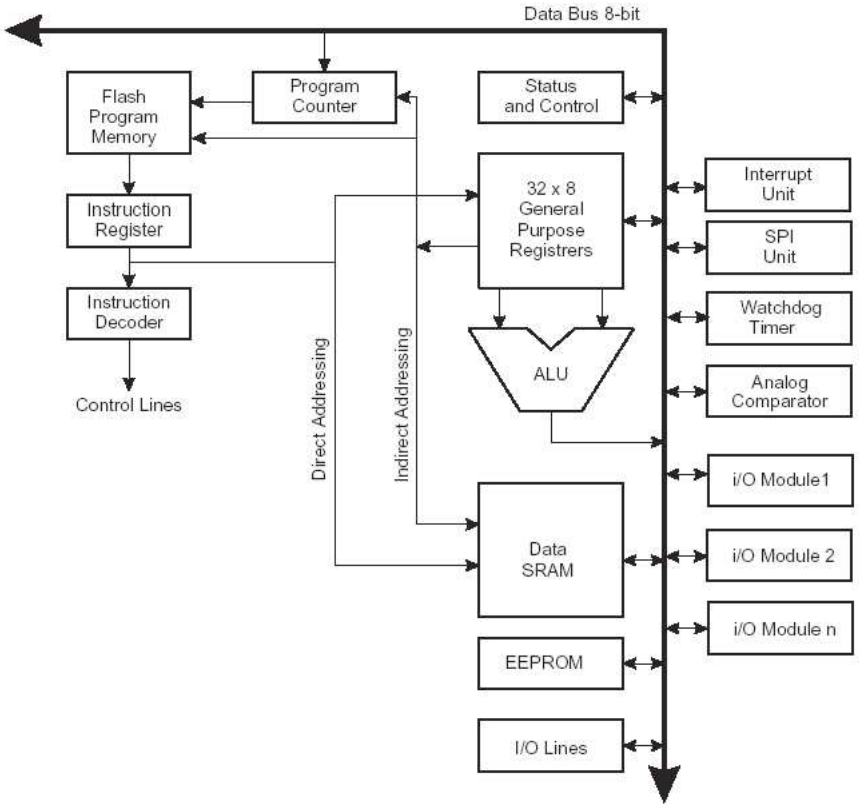
2.4 Microcontrollers

The garbage collection algorithm will be part of a Java Virtual Machine that runs on a microcontroller. Here follows a short description of microcontrollers in general. This information is gathered from [Dua98].

A microcontroller is a special component, dedicated to controlling an electronic device and can be found in devices like dishwashers, cars, CD/MP3-players and mobile telephones. Microcontrollers have a processor, internal memory, I/O modules, clock generator and timers (example block diagram in figure 2.6). These controllers are often designed to have a low power consumption because many of the devices that use microcontrollers are battery powered.

A microcontroller can often be placed directly in the device that it should control, hooked up to control lines, programmed and then function properly.

A general purpose microprocessor (found in most personal computers) cannot function without a large number of supporting components like external memory, mother board, hard drives, network controllers and so on. Microcontrollers are therefore more suited to be used in small electronic devices, called embedded systems.



The picture is copied from [Atm03].

Figure 2.6: Example block diagram of a microcontroller.

2.4.1 Java on a microcontroller

Java ability to "Compile once, run everywhere" makes Java a suitable platform for creating games and other applications for mobile telephones and other devices that use microcontrollers. The idea is that the producer of a Java program can create a program package and spread it to all its users by the Internet or some other means of communication.

The limited capabilities of the microcontroller, both regarding processing powers and memory size, makes it hard to run Java program efficiently. The main reasons for this is that the Java byte codes needs to be interpreted to be executed on must architectures. Garbage collection also reduces the execution. There have been a discussion recently about if Java is suitable for a microcontroller, like in [Nis05].

2.5 Atmel's 32-bit microcontroller

A short description of the microcontroller that the garbage collection algorithm is implemented for is presented here. This information is found in [Atm04a].

In appendix B the microcontroller white paper is included for readers interested in this new microcontroller. The white paper in this appendix is the property of Atmel and is not available for the general public, because it is still in the late stages of development.

At the time of this writing, the microcontroller is not yet available. The microcontroller is still in the late stages of development and Atmel has a working prototype that can run complex programs.

Atmel's new 32-bit microcontroller is designed for cost-sensitive embedded applications. It is designed to have a low power consumption and high code density.

Some key features:

- 32-bit load/store Reduced Instruction Set Computer (RISC) architecture.
- 15 general purpose registers.
- Unified memory model for easy access to the entire data and program memory space.
- Pipelined architecture.
- Branch prediction.
- Optional extensions for Java, SIMD and coprocessors.

To be able to run Java programs the microcontroller must have a Java Extension Module (JEM) that enables the microcontroller to run many of the Java bytecodes directly on the processor. Some bytecodes are too complex to be executed directly and these instructions are "trapped" and executed as a small RISC program. Below follows an example that shows how trapped bytecodes are handled.

Figure 2.7 shows a Java program running on the microcontroller. First the Virtual Machine is initialized. After the initialization is finished, a special `retj` instruction is executed. This instruction puts the microcontroller in Java mode, which enables execution of a Java program containing Java bytecodes. The microcontroller will now interpret the program as Java bytecodes instead of RISC instructions.

The Java program executes, but after a few steps a `invokestatic` command is executed. This command is too complex to be run in hardware and triggers a Java trap. The controller is set back to RISC mode and jumps to a predefined place in the main memory. This place contains a small RISC program that performs the `invokestatic` command. After the trap is completed the `retj` command is executed again. This takes the controller back into Java mode and the execution of the Java program continues. After the Java program is finished, the microcontroller returns to RISC mode.

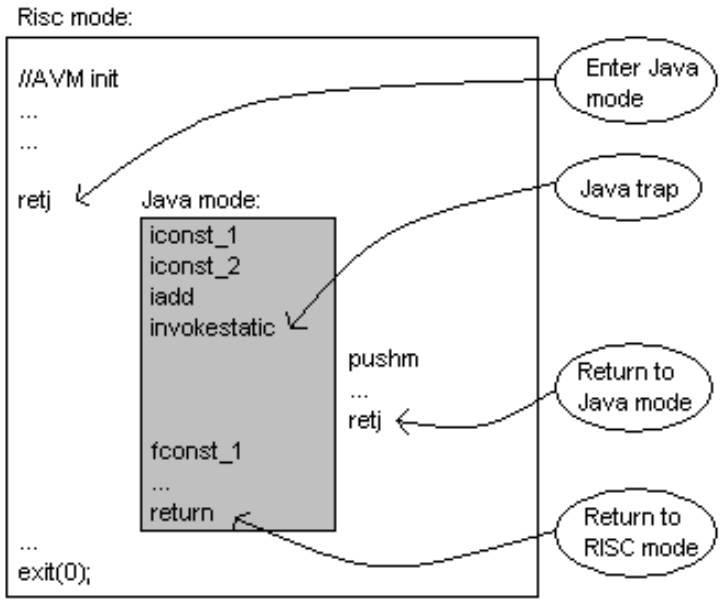


Figure 2.7: Java traps.

2.5.1 Atmel's Java Virtual Machine

In addition to the Java extension module, the microprocessor must have a Java Virtual Machine to comply with the Java Virtual Machine specification [LY99]. Atmel's Java Virtual Machine implementation is called the Atmel Virtual Machine (AVM).

Because the AVM is running on a microprocessor and is implementing the CLDC spec, a part of the verification process is taken out of the AVM and is performed by a preverifier (This is shown in figure 2.8). The preverifier makes the in-device verification faster by including some extra information in the class file.

A figure of the relationship between the AVM and the Java Extension Module (JEM) is shown in figure 2.9. The JEM contains the hardware support for the Java bytecodes. The JEM is also responsible for invoking the correct trap for the instructions not supported in hardware. Some Java traps perform operations on objects and must therefore have access to the heap where the objects are stored. Java traps are also used to invoke other methods and must therefore have access to the method area. The scheduler is responsible for letting every thread run once in a while and the garbage collector is responsible for managing the heap.

At the project start the AVM used an implementation of the mark-sweep garbage collection algorithm. This algorithm has a too high program interruption time to be used in combination with interactive programs. Tests done in the project leading to this diploma showed that the program interruption time could be greater than one second in some cases.

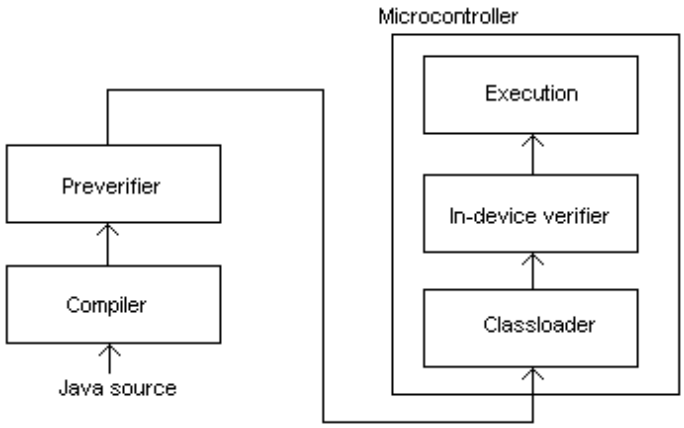
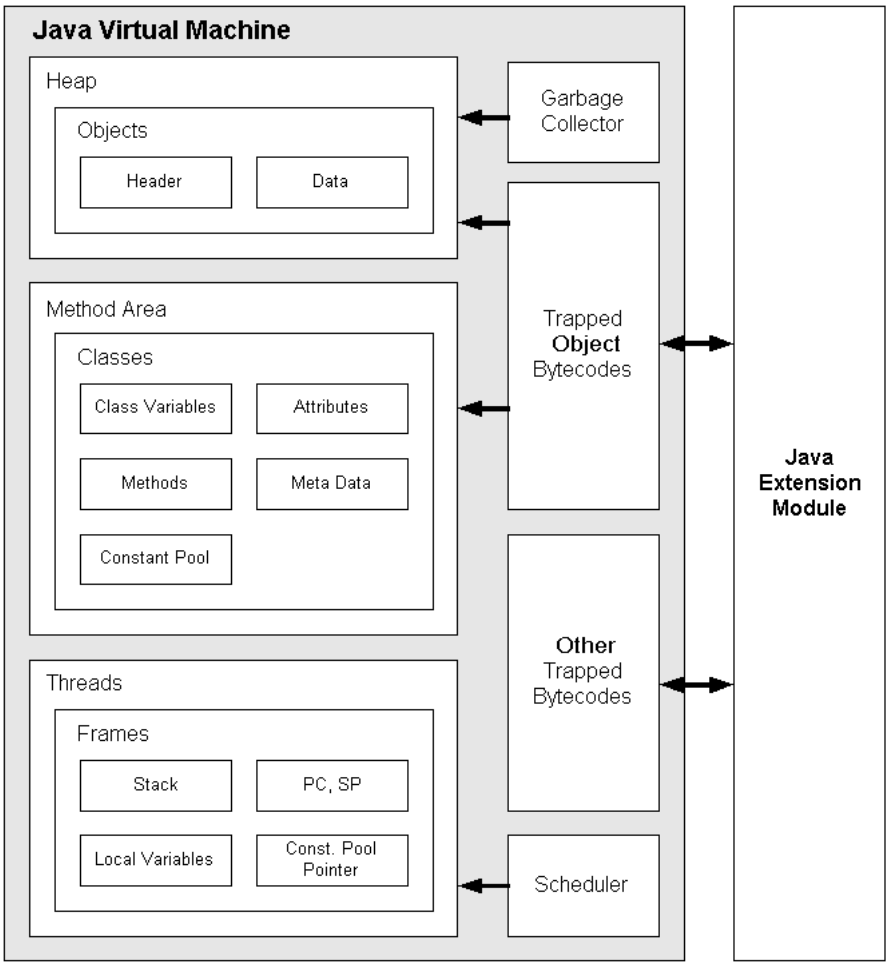


Figure 2.8: Compiling and running a Java program for the AVM.

2.6 Background summary

This chapter has described some terms from the project assignment and should make a reader, with little knowledge of the terms, better suited to understand this report.

The next chapter will contain a theoretical comparison of the implemented mark-sweep algorithm and the new MC^2 algorithm.



The grey area to the left represent the software parts of the Virtual Machine, while the white box to the right represents the hardware module. This figure is copied from [Atm04b].

Figure 2.9: The relationship between the AVM and the JEM.

Chapter 3

A theoretical comparison between the mark-sweep algorithm and the MC^2 algorithm

In the project leading to this diploma the mark-sweep algorithm was implemented in the AVM, despite MC^2 being the most suitable algorithm. The reason for this was the hardware changes needed and the complexity of the MC^2 algorithm [Amu04].

This chapter examines and compares the mark-sweep and the MC^2 algorithm to try to estimate their relative performance and to get a better understanding of the new algorithm before implementation.

3.1 A note about pseudo-code

Pseudo-code is a generic way to describe an algorithm without using the syntax of a programming language. A natural human understandable language (like English) is used instead. There exists no common standard for pseudo-code and people have different opinions about how it should be written.

The pseudo-code within this report follows the three standards of good pseudo-code from Brian Shelburne at the university of Wittenberg [Bri05]:

1. *Number each instruction. This is to enforce the notion of an ordered sequence of operations. Furthermore we introduce a dot notation (e.g. 3.1 come after 3, but before 4) to number subordinate operations for conditional and iterative operations.*
2. *Each instruction should be unambiguous (that is the computing agent, in this case the reader is capable of carrying out the instruction) and effectively computable (do-able).*

3. Completeness. Nothing is left out.[Bri05]

An indentation of subordinate instructions is used instead of the dot notation.

3.2 The mark-sweep algorithm

The mark-sweep algorithm implemented in the AVM is a version of the Deutch-Schorr-Waite pointer reversal algorithm modified by Thorelli [Tho72],[SW67]. The difference between this version and the original Thorelli modified Deutch-Schorr-Waite algorithm is that the implemented algorithm does not need a mark stack in the marking phase and therefore saves memory [Amu04].

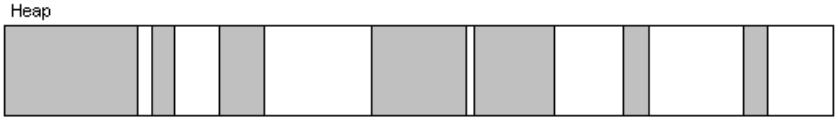
In this section the Deutch-Schorr-Waite (DSW) algorithm is examined, instead of the standard mark-sweep algorithm. By comparing the DSW with the MC^2 algorithm one can get a more realistic result because the DSW is the algorithm implemented in the AVM. Information about the standard mark-sweep algorithm can be found in [Jon96].

Pseudo-code for the mark-sweep algorithm can be found in algorithms 3.1, 3.2, 3.3, 3.4 and 3.5. Because this garbage collection algorithm is a modified version of an algorithm there exists no pseudo-code for it. The pseudo-code in this section was therefore written based on the source code for the actual implementation of the garbage collector.

This algorithm will also be explained in more detail in the following subsections.

3.2.1 Heap layout

The mark-sweep algorithm does not partition the heap up into different regions, but stores its objects on an uniform heap. An example mark-sweep heap is shown in figure 3.1.



Used memory is colored grey and free space is colored white.

Figure 3.1: Heap organization with the mark-sweep algorithm.

Because the mark-sweep algorithm does not compact the heap, fragmentation can become a problem. Fragmentation occurs when the algorithm reclaims unreachable memory between live objects. After a long time of execution the memory can consist of several small holes which no object fits into and thus wasting a lot of memory.

Figure 3.2 shows an example of fragmentation. In A) the heap contains four objects occupying the whole heap. B) shows the heap after object 2 has been reclaimed. Now there is 20 KB of free memory where object 2 used to be.

```
1  Algorithm:allocate(class)
   Comments:This function is called when a new object is allocated.
2  set freeListElement to first element of free memory block list
3  while size of freeListElement is less than size of instance of class do
4      next freeListElement
5  end
6  if no suitable freeListElements found then
7      gc()
8      set freeListElement to first element of free memory block list
9      while size of freeListElement is less than size of instance of class do
10         next freeListElement
11     end
12     if no suitable freeListElements found then
13         exit
14     end
15 end
16 make new instance of class at freeListElement
17 set objectPointer to point to the new object
18 add remaining space of freeListElement to freeList
19 return objectPointer
```

Algorithm 3.1: Mark-sweep: Pseudo-code for the allocate procedure.

Object 4 is reclaimed in C). If the system now wanted to allocate an object, 40 KB large, this request would be rejected, because no free memory block is larger than 20 KB.

3.2.2 Memory allocation

Because the free memory can occur in between objects, the mark-sweep garbage collector needs to keep a list of free memory to manage this free space. When space for an object is needed, this list is traversed by an insertion algorithm (Consult [Sta98] for details about insertion algorithms.) until a suitable block is found.

```
1  Algorithm:gc()
   Comments:This function is called from the allocate function when the
            heap is full, in intervals or explicitly from the user's Java
            program by the Java method System.gc().
2  for all root pointers do
3      mark(root)
4  end
5  sweep()
6  cleanup()
```

Algorithm 3.2: Mark-sweep: Pseudo-code for the gc procedure.

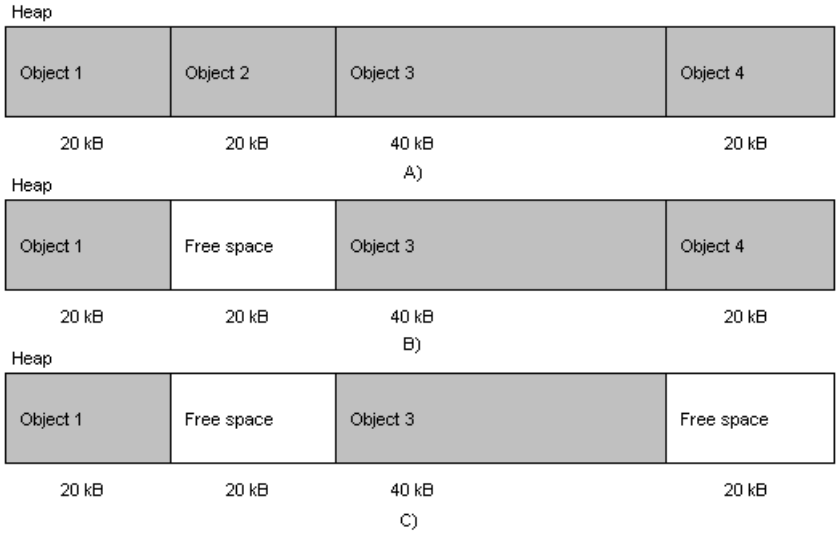


Figure 3.2: Fragmentation example.

If the insertion algorithm does not find a suitable block for the object to be allocated in, the garbage collector is invoked. When it finishes, the insertion algorithm is invoked again. If no free block exists that can hold the object is found, after garbage collecting, the JVM exits with an error code.

The pseudo-code for the mark-sweep algorithm's allocation procedure can be found in algorithm 3.1.

3.2.3 Garbage collection

The mark-sweep algorithm, consists of three phases (pseudo-code in algorithm 3.2):

1. The Mark phase.
2. The Sweep phase.
3. The Cleanup phase.

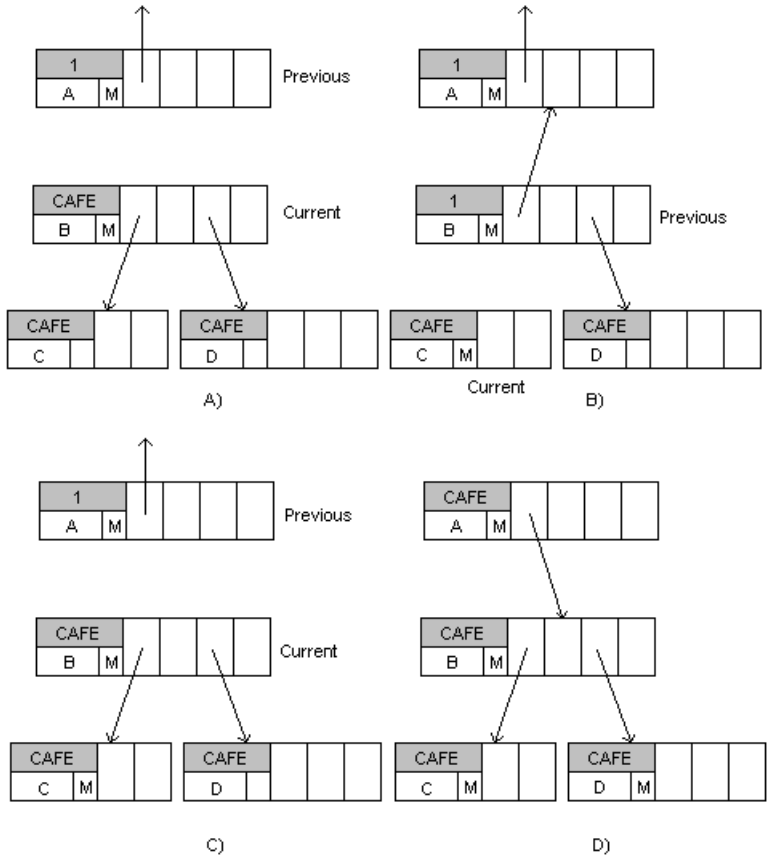
These phases will be described in detail in the next subsections.

The mark phase The mark phase is the most complex and important part of the garbage collection algorithm (pseudo-code in algorithm 3.3). Here lies the differences between the standard implementation and the pointer reversal algorithm implemented in the AVM.

The mark phase begins with discovering the garbage collection roots. In Java, the garbage collection roots are stored in static class fields and in each frame on the call stack. These roots are pointers that points into the heap, and

defines the source for the graph of live objects. All objects pointed to by the root pointers are then traversed with the pointer reversal algorithm.

For an object pointed to by a root pointer, all objects reachable by this object must be marked. When scanning an object, the garbage collector must mark and scan all objects the scanned object points to. Normally one would need to keep a stack of objects to keep track of which object to return to and how many pointers that have been scanned. In the pointer reversal algorithm, the mark stack is avoided by reversing the line of pointers (Normal pointers in the Java objects.). To keep track of which pointer that is reversed a counter is stored in the object header (This header normally just contain the word `0xCAFE`). An example with the pointer reversal algorithm is shown in figure 3.3.



Only pointers are shown in objects. An M in the object header symbolizes the Mark bit.

Figure 3.3: Pointer reversal example.

In subfigure A) of the example, the garbage collector is scanning object B. To reach object B, it has previously scanned object A (Pointed to by an object root.). Both objects A and B therefore have the mark bit set (indicated with an M in the header). Two pointers are kept, one that points to the current object (`current`) and one that points to the previous object (`previous`).

Observe that the topmost pointer in subfigure A) has been reversed and is

now pointing towards the object root. Notice that the header in object A also has been overwritten with a number. This indicates that it was the first pointer from Object A that was reversed when moving to object B. This information is necessary to rebuild the pointer graph when retreating from an object.

In the next subfigure the collector starts scanning object C, pointed to by object B. Before leaving object B it reverses the pointer originally pointing to object C. This pointer now points to the previous object, A. Object C is marked after changing the pointer.

In subfigure C) the garbage collector have finished marking object C and have reversed the pointers again, so that the pointer to object C is restored. The `current` pointer now points to object B and the `previous` pointer points to object A. In subfigure D) the collector have finished marking. The pointer originally pointing to A has also been restored. The object tree is now restored to its original form and all objects in the tree have been marked. All object headers are also restored.

The sweep phase During the sweep phase the garbage collector scans the heap and inserts the garbage objects (The ones with the mark bit set to zero) into the list of free memory. Pseudo-code can be found in algorithm 3.4.

The cleanup phase The cleanup phase (see algorithm 3.5) scans through every remaining object and clears the mark flag to prepare for the next garbage collection cycle.

3.3 The Memory-Constrained Copying algorithm

The Memory-Constrained Copying algorithm is a special garbage collection algorithm designed for embedded devices and was developed by Sachindran, Moss and Berger at the university of Massachusetts [SMB04].

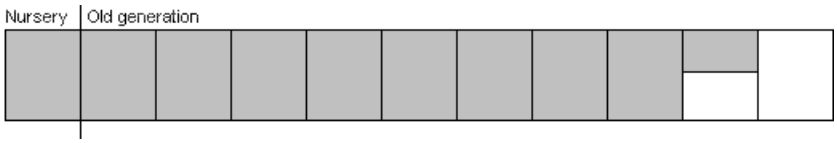
A pseudo-code for the Memory-Constrained Copying algorithm can be found in algorithms 3.6, 3.7, 3.8, 3.9, 3.10 and 3.11. Except for the `nurseryMark()` procedure. This procedure is not specified in the MC^2 paper and can be chosen when implementing the algorithm. The MC^2 paper did not include a pseudo-code of the algorithm (except the write barrier) so the following pseudo-code is written based on the description of the algorithm in the MC^2 paper [SMB04].

A textual description of the MC^2 algorithm can also be found in the following subsections.

3.3.1 Heap layout

The MC^2 heap consists of several equal sized windows. One window holds the newly created objects and is called the nursery. The older objects occupies the rest of the windows (the old generation windows).

Figure 3.4 shows the MC^2 heap organization. The nursery section is the window to the left, the rest belongs to the old generation.



Used memory is colored grey and free space is colored white.

Figure 3.4: Heap organization with the MC^2 algorithm.

3.3.2 Memory allocation

Memory is allocated from the nursery section of the heap. The objects in the nursery is allocated serially from one end of the nursery to another. This makes allocation of object with the MC^2 algorithm fast. See pseudo-code in algorithm 3.7.

3.3.3 Garbage collection

Nursery collection (see pseudo-code in algorithm 3.8) happens when the nursery is running full. The nursery is marked and the marked objects are copied to the old generation in one sweep. Because the nursery is so small, the program interruption is kept to a minimum.

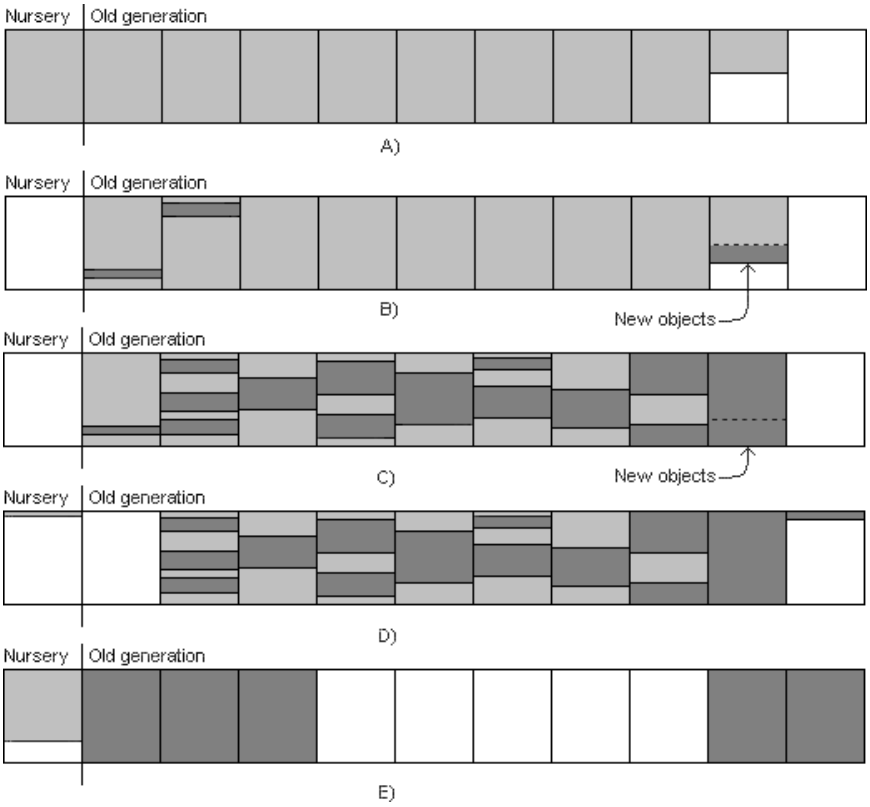
After every nursery collection, the collector checks the old generation. If the old generation is more than 80 percent full it triggers an old generation marking. This marking is not performed at once, but incrementally: A little bit of the heap is marked for every nursery allocation (see algorithm 3.9). The collector calculates how much memory it will need to mark to finish marking before the old generation has only one empty window left. If this threshold is reached, the collector stops program execution and marks the rest of the old generation.

During marking the nursery section can run full with new objects and the nursery must be collected. All live objects from the nursery are then collected and put in the old generation, with the mark bit set.

When the whole heap is marked, the collector enters the copying state. With every following nursery allocation, a little bit of old generation copying is "piggybacked" to the allocation. In the copying phase all live objects are copied to a new window, except windows with a large number of live objects (e.g. 95% live objects). These windows are just skipped because the gain in memory is not worth the time required to copy all the live objects. A pseudo-code for the old generation marking procedure can be found in algorithm 3.10.

After all live objects have been transferred from a window, the window is marked as empty and can be filled with new objects. The copying phase continues until all the objects in the old generation are copied.

An example of a garbage collection cycle with MC^2 is shown in figure 3.5. Subfigure A) shows the initial layout, right after the old generation reached the 80 % occupation mark. To the left is the single nursery window and to its right are the old generation windows. Because the threshold has been reached, the garbage collector starts an old generation marking.



Allocated memory is colored grey, marked objects are colored dark grey and unallocated memory is white.

Figure 3.5: MC^2 garbage collection example.

In subfigure B) a nursery collection has taken place (observe the new marked objects in the old generation) and some marking have also been done during nursery allocation. The objects copied to the old generation from the nursery are automatically marked because they was alive during the nursery collection. In subfigure C) the whole old generation is marked and there is only one free window left. This triggers the copying phase.

The copying phase is carried out during nursery allocation. In subfigure D) the first object has been allocated in the nursery and the live objects from the first old generation window have been copied to the last old generation window. The unmarked (unreachable data) is not copied to the new windows. These objects are just left in the window and eventually overwritten by new objects.

In the last subfigure, all the live data in the old generation is copied to new windows, discarding unreachable data. Nursery collection can now continue as normal until the old generation reaches the 80% threshold again and triggers marking. Observe that window number ten, counting from left to right, has not been copied. This is because it is a high occupancy window and was therefore not copied during this phase.

3.4 Comparison

Atmel's microcontroller does not have many spare resources when running Java code, concerning both processing power and main memory. It is therefore vital that the garbage collection algorithm consumes as little of these resources as possible.

The algorithms are compared with respect to four measures: memory utilization, program interruption, throughput and algorithmic complexity. Because Atmel expects this microcontroller to run mostly interactive Java applications the most critical measure is the program interruption time, followed by the throughput and at last the memory utilization and the algorithmic complexity. These priorities stem from [Amu04].

3.4.1 Memory utilization

Some garbage collection algorithms have heap layouts that require them to reserve some space for permanent structures managing the heap. Other algorithms can have a need to reserve some part of the heap to make it possible to move around objects. In a microcontroller these structures can "waste" valuable heap space that could have been used to store objects.

The mark-sweep algorithm treats the whole heap as a single unit and does therefore not need any data structures to manage the heap. Because it does not move objects around either, it does not need to reserve storage to copy the objects into and can utilize the heap to its maximum. The mark-sweep algorithm suffers from fragmentation because it never compacts the heap. Fragmentation could lead to that the algorithm must terminate the Java Virtual Machine because it cannot find a suitable room for an object on the heap.

The MC^2 algorithm organizes its heap space in equal sized windows, managed by a data structure. This will waste some heap space in each window. The algorithm will stop user program execution and run the garbage collection until completion if only one completely free window exists, in the marking phase, and there is not enough room to allocate new objects in the other windows. One window is therefore at any time available on the heap. The reason for keeping one window free is that in the copying phase there must exist one free window to copy the objects into. This algorithm does not suffer from fragmentation on an object level, but the window organization of the heap can lead to fragmentation. The MC^2 algorithm can, in contrast to the mark-sweep algorithm, move objects around and therefore counter this problem.

The MC^2 algorithm therefore utilizes the heap memory a little less efficiently than the mark-sweep algorithm. The amount of memory wasted is depending of the window size and the size of the data structures. For a heap with e.g. one nursery and 10 old generation windows the MC^2 will be able use around ten percent less memory for objects. Fragmentation is a problem for both algorithms, but the MC^2 algorithm has an advantage because it can overcome this fragmentation by compacting the heap.

3.4.2 Program interruption

Program interruption occurs when the focus of the Central Processing Unit (CPU) shifts from the user program to another process, like garbage collection. If these program interruptions are very short the user would not notice, but longer program interruptions can be very frustrating. Because the AVM is designed to be used with interactive programs, like games, program interruption is important.

The mark-sweep algorithm is a typical "stop the world" algorithm. When garbage collection is triggered it runs until completion, totally regardless of what other programs that may be running. When the garbage collector encounters many small, interlinked objects, like in a binary tree, the marking period may take a substantial amount of time.

The MC^2 algorithm is designed to have a short program interruption time. The incremental design of this algorithm makes it possible for the normal program execution to run along with the garbage collector.

The figure 3.6 shows an example of how large the program interruption can be with the two algorithms. When the mark-sweep garbage collector starts, it occupies all available CPU time, leaving no time to the user program.

The MC^2 algorithm starts its marking when the heap is 80% full. Normal programs still allocate memory when the garbage collector marks the heap. This allocation happens more slowly than in the mark-sweep example because the garbage collector runs in the background. The copy phase also lets the user program continue as normal and therefore the total garbage collection time for MC^2 is longer than for mark-sweep.

This means that the program interruption of the MC^2 is significantly smaller than the mark-sweep interruption. The MC^2 article confirms this:

> *"The pause times for MC^2 is 10-17 times lower than a copying garbage collector and 7-13 times lower than mark-sweep in a heap that is 1.8 times the program live size[SMB04]."*

3.4.3 Throughput

The throughput of the algorithms have an impact on how long time it takes to execute a program. Although this does not have a great impact for an interactive program, CPU intensive tasks, like image decoding, are more affected.

The mark-sweep algorithm only marks live objects and attaches unreachable objects to a free list. It does not therefore create much overhead, when compared to an algorithm that must copy every live object from one place to another.

The MC^2 algorithm has a much greater overhead than the mark-sweep algorithm when garbage collecting. This is mainly because it has to copy every live object from one window to another in the copy phase. Memory allocation is much faster because the nursery is managed like a stack, in contrast to the free list from the mark-sweep algorithm.

The MC^2 article reports that the throughput of the algorithm closely matches the throughput of the mark-sweep algorithm:

> *"We compared the performance of MC^2 with a non-incremental generational mark-sweep collector and a generational mark-compact collector, and showed that MC^2 provides throughput comparable to that of both of those collectors [SMB04]."*

This could indicate that the loss of throughput in copying is covered by the gain in memory allocation for the MC^2 algorithm.

3.4.4 Algorithmic complexity

The algorithmic complexity is a measure of how complex the algorithm is and therefore how difficult it is to implement and maintain the algorithm. This is also called the essential complexity, the complexity in an implementation that you cannot remove (In opposition to the accidental complexity, which is caused by the programmer while implementing the algorithm.) [McC93].

The algorithmic complexity of the garbage collection algorithm is directly cost related. Algorithmic complexity means that it is more difficult to maintain the implementation and it would require more time and concentration.

The mark-sweep algorithm's essential complexity is rather low. The garbage collector is invoked, from the object allocation code, each time the heap runs full. The algorithm then marks the heap, sweeps it and then returns to the Java program again.

The MC^2 algorithm has a more complex behavior. It is invoked from the object allocation code, like the mark-sweep algorithm, but it is also invoked when the write barrier is triggered. The MC^2 algorithm deals with a more complex heap structure and can be in several states, like old generation marking, old generation copying and normal state.

The MC^2 algorithm is therefore a more complex algorithm than mark-sweep. This could be a problem while implementing and debugging the MC^2 algorithm and one should therefore be very careful not to include accidental complexity in the MC^2 implementation also.

3.5 Comparison summary

The comparison (Summary in table 3.1.) shows that the critical program interruption should be much smaller with the MC^2 than with the mark-sweep. The total execution time should be around equal and the memory utilization should be a bit smaller with the MC^2 algorithm. The MC^2 algorithm has a much higher algorithmic complexity than the mark-sweep algorithm.

The low program interruption of the MC^2 algorithm looks very promising and encourages the implementation of the algorithm, especially because it is a high priority requirement for the garbage collector. The next chapter contains a short introduction to the Java memory system in the microcontroller and the changes performed on the AVM before implementing the MC^2 algorithm.

```
1  Algorithm:Mark(Root)
   Comments:This function is called from the gc function to mark an object
            root on the heap.
2  set currentObject to Root
3  set previousObject to NIL
4  while currentObject not equals NIL do
5      if currentObject not marked then
6          set mark bit in currentObject
7      end
8      read counter from header of currentObject
9      if counter equals 0xCAFE then
10         set counter to 0
11     end
12     while counter is less than number of fields in currentObject and no
       pointers are found do
13         if currentObject[counter] is an object pointer and this pointer
           points to an unmarked object then
14             pointer is found
15         end
16         increase counter with 1
17     end
18     if pointer is found then
19         store counter in currentObject's header
20         store pointer to previousObject at currentObject[counter - 1]
21         set previousObject to currentObject
22         set currentObject to the object pointed to by the pointer found
23     else
24         set tempObject to currentObject
25         set header of currentObject to 0xCAFE
26         set currentObject to value of previousObject
27         load counter from currentObject's header
28         set previousObject to currentObject[counter - 1]
29         set currentObject[counter - 1] to tempObject
30     end
31 end
```

Algorithm 3.3: Mark-sweep: Pseudo-code for the mark procedure.

```
1  Algorithm:sweep()
   Comments:This function is called from the gc function after the heap has
            been marked.
2  for all objects on heap do
3      if object is not marked then
4          insert object into free list
5      end
6  end
```

Algorithm 3.4: Mark-sweep: Pseudo-code for the sweep procedure.

```
1  Algorithm:cleanup()
   Comments:This function is called from the gc function to clean up and
            prepare for the next garbage collection.
2  for all objects on heap do
3      clear mark bit in object
4  end
```

Algorithm 3.5: Mark-sweep: Pseudo-code for the cleanup procedure.

```
1  Algorithm:writeBarrierTriggered(sourceObject, targetObject)
   Comments:This function is called when the write barrier is violated. The
            pseudo-code for this function was copied from [SMB04]
2  if sourceObject is not in nursery then
3      if targetObject is in nursery then
4          set nurseryMark in targetObject
5      else
6          if state is oldGenerationMarkingState then
7              if sourceObject is in old generation and mark bit is set in
               sourceObject then
8                  if mark bit is not set in targetObject then
9                      push targetObject on mark stack
10                 end
11             end
12         end
13     end
14 end
```

Algorithm 3.6: MC^2: Pseudo-code for the writeBarrierTriggered function.

```
1  Algorithm:allocate(class)
   Comments:This function is called when a new object is allocated.
2  if free space in nursery is less than size of instance of class then
3      nurseryCollect()
4  end
5  push a new instance of class on nursery
6  set objectPointer to point to new instance
7  return objectPointer
```

Algorithm 3.7: MC^2: Pseudo-code for the allocate function.

```
1  Algorithm:nurseryCollect()
   Comments:This function is called from the allocate function when the
            nursery is full.
2  if state equals oldGenerationCopyingState then
3      oldGenerationCopy()
4  end
5  if state equals oldGenerationMarkingState then
6      oldGenerationMark()
7  end
8  for every root pointer in nursery do
9      nurseryMark(root object)
10 end
11 for every marked object in nursery do
12     find free oldGenerationWindow
13     copy object to oldGenerationWindow
14 end
15 reset(nursery)
16 if oldGenerationMarkThreshold reached then
17     set state to oldGenerationMarkingState
18 end
```

Algorithm 3.8: MC^2: Pseudo-code for the nurseryCollect function.

```
1  Algorithm:oldGenerationMark()
   Comments:This function is called from the nursery collector when an old
            generation marking increment should be done.
2  for every root pointer do
3      if object pointed to by root is not marked then
4          push root on mark stack
5      end
6  end
7  while mark stack not empty and mark threshold not reached do
8      set targetObject to object popped from mark stack
9      set mark bit in targetObject
10     for all objects pointed to by targetObject do
11         if object is not marked then
12             push object on mark stack
13         end
14     end
15 end
16 if mark stack is empty then
17     while there exists a window that is not in a group do
18         find a set of windows that together can fill up one free window
           with objects
19         Add these windows to a group
20     end
21     set state to oldGenerationCopyingState
22 end
```

Algorithm 3.9: MC^2: Pseudo-code for the oldGenerationMark function.

```
1  Algorithm:oldGenerationCopy()
   Comments:This function is called from the nursery collector when an old
            generation copying increment should be done.
2  set windowGroup to first window group
3  set targetWindow to first free window
4  for all windows in windowGroup do
5      for all live objects in window do
6          copy object to targetWindow
7      end
8  end
9  if no more windowGroups then
10     set state to normalState
11 end
```

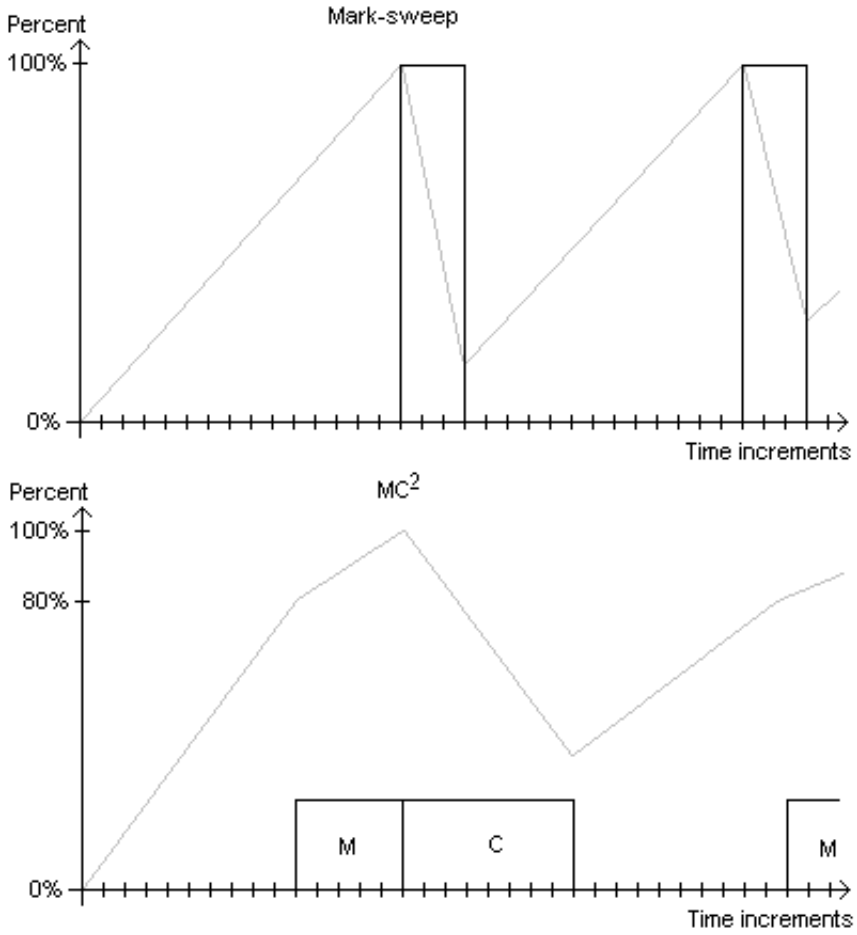
Algorithm 3.10: MC^2: Pseudo-code for the oldGenerationCopy function.

```
1  Algorithm:gc()
   Comments:This function is called explicitly by the user's Java program
            by the Java method System.gc().
2  while state equals oldGenerationMarkingState do
3      oldGenerationMark()
4  end
5  while state equals oldGenerationCopyingState do
6      oldGenerationCopy()
7  end
8  nurseryCollect()
9  set state to oldGenerationMarkingState
10 while state equals oldGenerationMarkingState do
11     oldGenerationMark()
12 end
13 while state equals oldGenerationCopyingState do
14     oldGenerationCopy()
15 end
```

Algorithm 3.11: MC^2: Pseudo-code for the gc function.

Measure	Priority	Mark-sweep vs. MC^2	Comment
Program interruption	H	MC^2 best	Incremental marking lowers program interruption.
Throughput	M	Equal performance	The loss from the more complex GC is gained by the more efficient allocation.
Memory utilization	L	Equal performance	MC^2 must always have one free window, wasting some space. MC^2 can deal with fragmentation.
Algorithmic complexity	L	Mark-sweep best	MC^2 more complex.

Table 3.1: Summary of the comparison between the mark-sweep and the MC^2 algorithm.



The grey line shows how full the heap is and the black line shows the garbage collection CPU time in each time increment. The M in the MC^2 garbage collection marks the marking phase of the collection and the C marks the copying phase.

Figure 3.6: Program interruption comparison.

Chapter 4

Preparations before implementing the MC^2 algorithm

First in this chapter is a little introduction to the Java memory interface for readers unfamiliar with the AVM. This chapter also includes a description of the modifications performed on the AVM before the MC^2 algorithm could be implemented.

4.1 Java memory interface

This short introduction covers the object and array structure, the object handles and special Java instructions designed to assist the garbage collector.

4.1.1 Objects and arrays

The object and array structure at the project start is shown in figure 4.1. Objects and arrays consists of a header and data. The header is all fields below the stippled line in the figure.

A description of the object header items is found in table 4.1 and array header items are found in table 4.2.

4.1.2 Object handles

The handle bit is a part of the status register in the microcontroller. If the handle bit is not set all object references (including arrays references) are direct pointers. If the handle bit is set, the references are indirect. This is shown in figure 4.2.

Because objects are word aligned, the two least significant bits of an object address will always be zero. To assist the garbage collector it was decided that

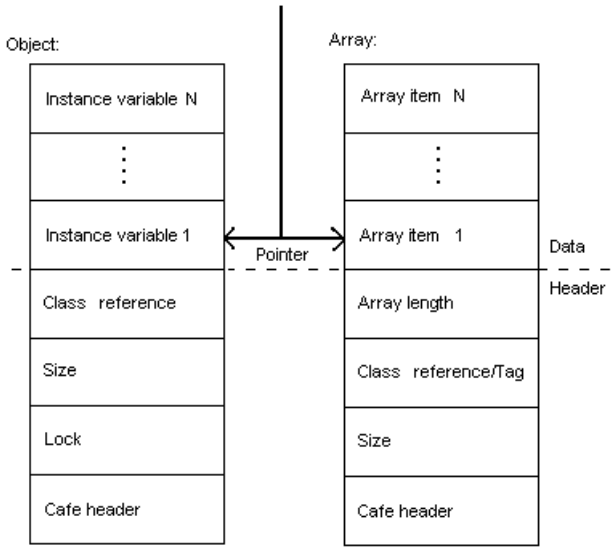


Figure 4.1: Object and array structure.

Field	Description
Cafe header	The word `0xCAFE`.
Lock	The lock data structure associated with this object. When using Java threads a thread can lock an object and make it impossible to access it by other threads.
Size	The total size of the object.
Class reference	A reference to the class of the object.

Table 4.1: The object header items.

object can only be placed in the user space at addresses lower than `0x4000 0000`. This design makes the two most significant bits in the handle zero.

This makes it possible to set status bits for every object. The address of the real object is obtained by masking away the two most significant and two least significant bits when accessing an handle. These status bits are shown in figure 4.3 and a short description of these can be found in table 4.3.

To illustrate how the handles work two examples are provided below. One that does not use handle and one that does. These examples show how a field in an object is stored using the `putfield` instruction. The `putfield` instruction takes two operands, one pointer to the object to store the field in and the value to store. In addition to the operands, the `putfield` instruction is followed by a two byte index in the Java bytecode. This index is used as an offset to the object address, when storing the field.

Figure 4.4 shows the code for the implementation of the `putfield` instruction. In the code `objectref` is the second element on the operand stack and `value` is the first element. The first line checks if the `objectref` is a null pointer and throws a `NullPointerException` if it is. The line `if(SR(H))` checks if the

Field	Description
Cafe header	The word `0xCAFE`.
Size	The total size of the array, including the header.
Class reference/Tag	A class reference if it is an array of objects or a tag describing an array of primitive types (e.g. integers).
Array length	The number of items in the array.

Table 4.2: The array header items.

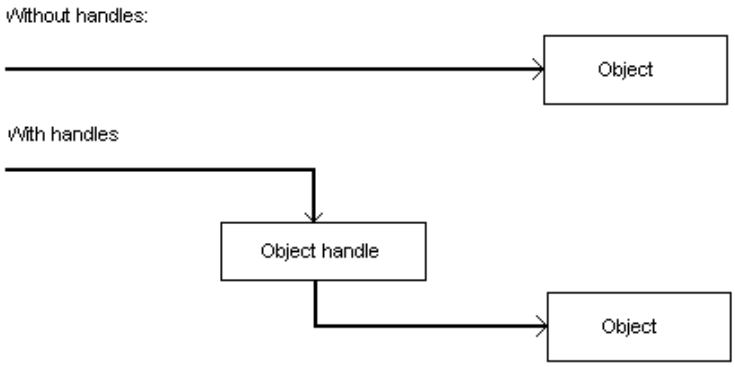


Figure 4.2: Object handles.

handle bit is set in the status register. If it is the two next lines accesses the object handle and masks away the status bits. This address is stored in the temporary value `addr`. If the handle bit is not set the `objectref` is stored directly in `addr`. After this the index bytes are added to the `addr` value and the `value` is stored at this location. The last line decreases the Java Operand Stack Pointer (JOSP) with two. This pops the two operand elements of the stack.

Example without handles

This example shows how fields are stored in an object without using object handles.

In figure 4.5 an example of the `putfield` command without using object handles is shown. In subfigure A) the Java bytecodes are shown to the left. The

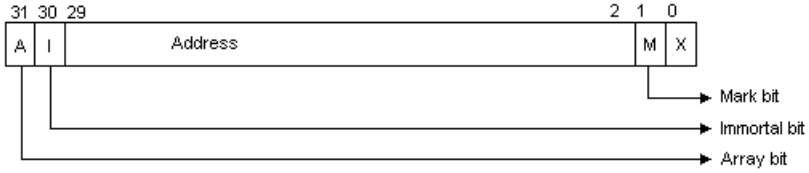


Figure 4.3: Object handle with array, immortal and mark bit.

Bit	Description
A	The array bit is set to one if the address field in the handle points to an array.
I	The immortal bit tells if the object should be collected by the garbage collector or not.
M	When the garbage collector marks an object it sets the mark bit to one.
X	Unused.

Table 4.3: The object handle status bits.

```
Instruction format:
putfield(0xB5), indexbyte1, indexbyte2

Operand stack:
..., objectref, value -> ...

if (objectref == NULL)
  throw NullPointer exception;
if(SR(H))
  R11 <- *objectref;
  addr <- (R11 & 0x3FFF FFFC);
else
  addr <- objectref;

*(addr + ((indexbyte1 << 8 | indexbyte2) << 2)) <- value;

JOSP <- JOSP - 2;
```

This code is copied from [Atm04a].

Figure 4.4: Code for the `putfield` instruction.

program counter (PC in the figure) is pointing to the `putfield` opcode. On the Java operand stack the value to be stored (`0x0000 0042`) is on top of the stack. Underneath this is the reference to the object to store the value into. These elements have been put on the stack by the `aload_0` and `iload_1` bytecodes found over the `putfield` bytecode. At the right side of the figure a bit of the main memory is shown, containing the object pointed to by the object reference on the stack. This object has four fields all containing the value `0x0000 0000`.

In subfigure B) the `putfield` instruction has been carried out. The program counter has advanced to the next instruction and the two operands have been popped of the stack. The value `0x0000 0042` has been stored in field two of the object, because of the `0x0002` offset after the `putfield` instruction in the bytecode.

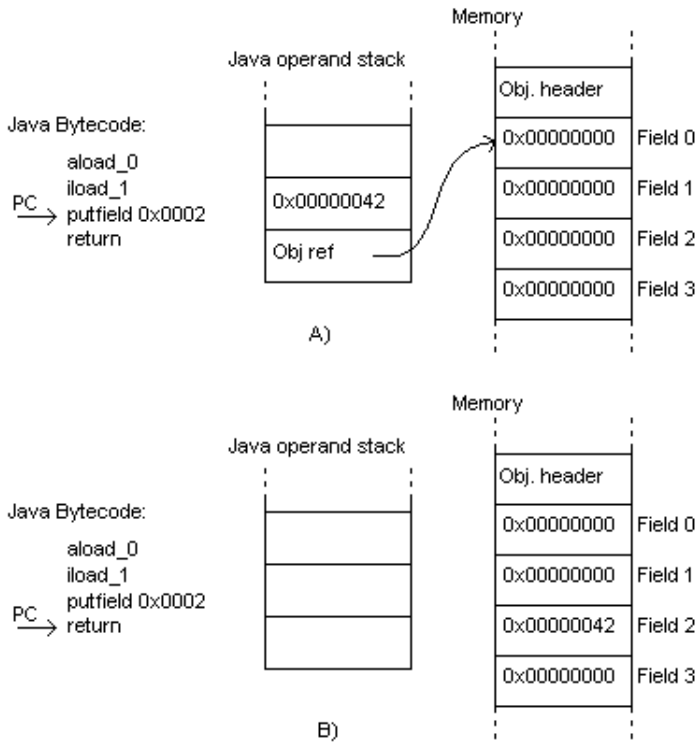


Figure 4.5: `putfield` example without handles.

Example with handles

This example shows how the `putfield` instruction is carried out when object handles are enabled.

Figure 4.6 shows the same Java bytecode as in the previous example, but now object handles are enabled. In subfigure A) the object reference on the Java operand stack points to an object handle in the main memory, instead of pointing directly to the object. The handle contains the address to the object in addition to some special status bits (These status bits are not shown.).

In subfigure B) the field has been stored in the object. To access the object the object handle pointed to by the object reference must be read. This value must be masked with the value `0x3FFF FFFC` before the real address of the object appears.

4.1.3 Special Java instructions

The MC^2 article [SMB04] mentions that the incremental nature of the marking can cause trouble when pointers between objects change. This is called a pointer mutation. It is therefore necessary to record pointer mutations to counter this

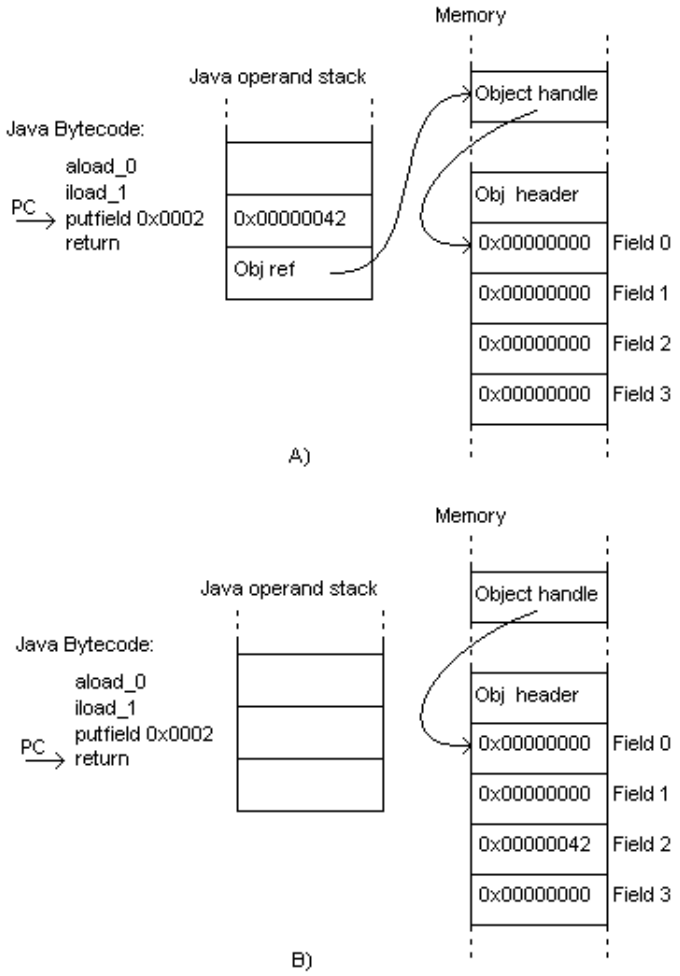
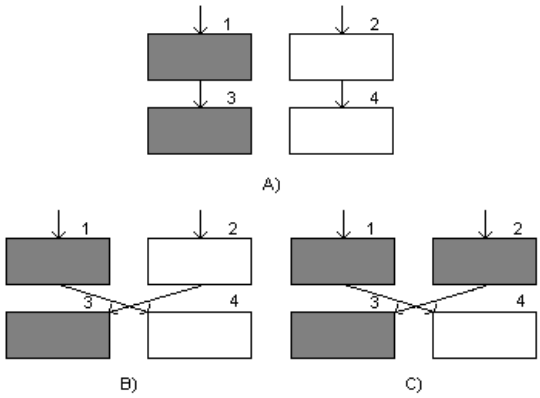


Figure 4.6: `putfield` example with handles.

error source. The following example (Figure 4.7) shows what can happen if pointer writes are not recorded.

In subfigure A) both objects 1 and 3 are marked during one increment of the marking phase. Normal program execution continues and at some point (subfigure B) in the figure) the pointers in objects 1 and 2 are swapped, making object 1 point to an unmarked object, object 3. In subfigure C) the marking continues, but object 4 is never marked because all pointers from object 1 was scanned in subfigure A) and the marker does not know that the pointer in object 1 has changed. Object 1 now has the mark bit set and is therefore never rescanned. Object 4 is then, falsely, reclaimed as garbage in the copy phase, leaving a dangling pointer behind.

To speed up nursery marking the MC^2 paper suggests that one should also record storing of pointers pointing into the nursery. If such pointers are not recorded the nursery collector must scan through the whole heap to find all



This figure is copied from [SMB04].

Figure 4.7: Error during incremental marking.

pointers pointing into the nursery, slowing down the collection enormously. For example if there is one nursery and ten old generation windows. The number of handles traversed is ten times greater if one does not record nursery pointers. This assumes a even distribution of handles among the windows.

```
writebarrier(sourceObject, sourceSlot, targetObject){
  if(sourceObject not in nursery){
    if(targetObject in nursery)
      record sourceSlot in nursery remset
    else if(targetObject in old generation){
      if(sourceObject is not mutated){
        set mutated bit in sourceObject header
        record sourceObject in mutated object list
      }
    }
  }
}
```

This code was found in [SMB04].

Figure 4.8: code for the MC^2 write barrier.

In figure 4.8 the code for the MC^2 write barrier is shown. The write barrier takes care of both storing pointer mutations in marked objects and storing nursery pointers. Because the write barrier only records storing of pointers to objects and arrays only three bytecodes are affected. These bytecodes are described in table 4.4.

Because these instructions are executed directly by the CPU the microcontroller hardware had to be modified to support the MC^2 algorithm. In the four sections below the hardware changes made in the microcontroller is described.

Bytecode	Description
`putfield`	This bytecode stores a reference in an object. It is also used to store integers, floats and other word-sized variables in an object.
`aastore`	This bytecode stores a pointer in an array of pointers.
`putstatic`	This instruction is used to store references, integers and floats to a class in a static context.

Table 4.4: Bytecodes that stores object pointers.

Java Barrier Configuration Register

The Java Barrier Configuration Register (JBCR) stores a pointer to the boundary between the old generation and the nursery. This makes it possible for the hardware to check if an object lies in the nursery or in the old generation.

To avoid having a separate enable bit for the write barrier, the write barrier can be disabled by inserting a value greater than `0x3FFF FFFC`. The masking of the status bits in the object handle forces objects to be stored in addresses lower than `0x3FFF FFFC`.

putfield

The `putfield` bytecode is used to store integers, floating point and other variables in addition to object references. There is no easy way for the CPU to know which type the value is at runtime, so every `putfield` can be a pointer store.

During linking, the class loader have access to information about what type is stored by the `putfield` command. This information is used to change the bytecode into either `putfield_quick` or `putfield2_quick`. To be able to only test for write barrier violation when storing object pointers, a new Java instruction is programmed into the microcontroller. This command is called `aputfield_quick`. The linker must be modified (in software) to substitute `putfield` with `aputfield_quick` when storing a pointer in the object.

The MC^2 write barrier pseudo-code in figure 4.8 tells that only pointer stores to objects in the old generation need to be trapped. The boundary between the nursery and the old generation, stored in the JBCR, can be used by the CPU to check if the pointer's value is in the old generation. If it is, the instruction will be trapped and carried out in software. This reduces the number of trapped instruction and increases execution speed of the `aputfield_quick` command.

aastore

The `aastore` command stores an object reference in a pointer array. This pointer store must also be checked for write barrier violation in the same manner as `aputfield_quick`.

This will only require changes in hardware because the `aastore` bytecode only is used to store pointers and therefore does not need to be treated differently in the linker.

putstatic

The `putstatic` bytecode can be handled entirely in software. The reason for always processing `putstatic` in software is that every store to the static fields of a class is a change of the garbage collection root pointers. These root pointers changes must always be processed by the garbage collector and is therefore handled in software.

Write barrier example

The following example shows the write barrier functionality.

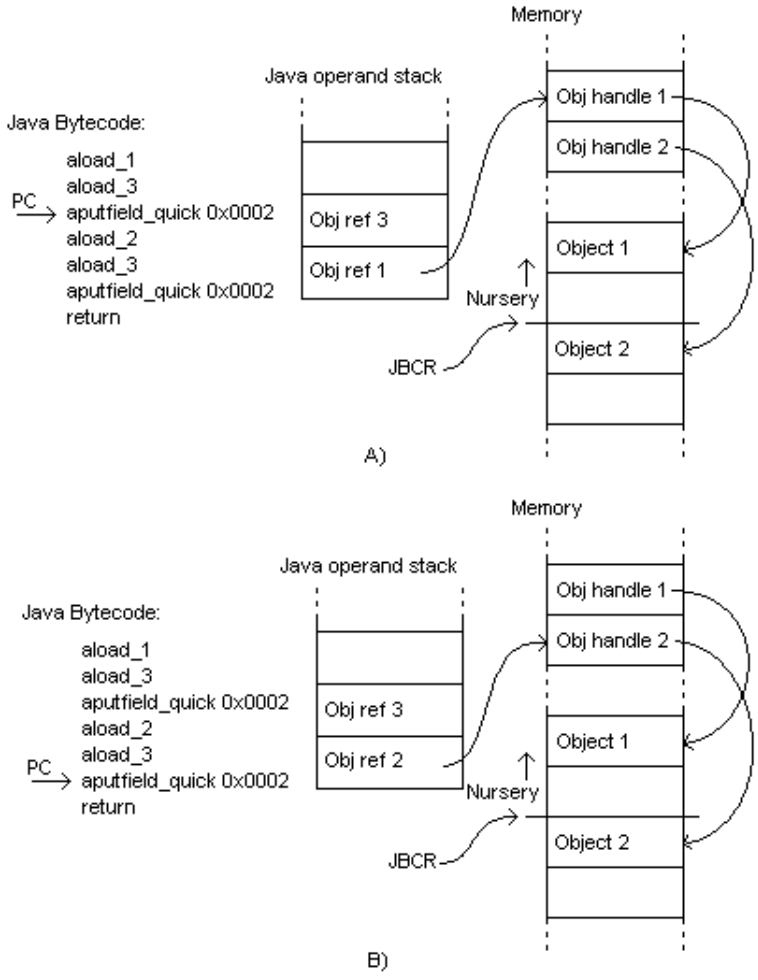


Figure 4.9: `aputfield_quick` example with write barrier.

In figure 4.9 A) the small Java code, on the left side, has been executed until the `aputfield_quick` instruction was reached. There are now two elements on the Java operand stack. The topmost item is the element to store in the object and the object on the bottom is the object to store the element in. The JBCR points to the border between the old generation and the nursery.

The object reference on the bottom of the stack points to a handle in the main memory. This handle points to object 1, which lies in the nursery section of the Java heap. when the `aputfield_quick` command is reached the CPU will check if the pointer in the handle to the object is greater than the JBCR register. If it is the CPU will jump to a specific RISC program and execute it. This can be seen in the `aputfield_quick` code in figure 4.10 on line 5 and 6. The rest of the code is equal to the code for the `putfield_quick` command.

```
Instruction format:
aputfield_quick(0xE7), indexbyte1, indexbyte2

Operand stack:
..., objectref, value -> ...

if(objectref == NULL)
  throw NullPointer exception;
if(SR(H))
  R11 <- *objectref;
  if ((R11 & 0x3FFF FFFC) >= JBCR)
    TRAP 0;
  addr <- (R11 & 0x3fff fffc);
else
  addr <- objectref;

*(addr + ((indexbyte1 << 8 |indexbyte2) << 2)) <- value;

JOSP <- JOSP - 2;
```

Figure 4.10: Code for the `aputfield_quick` instruction.

When the first `aputfield_quick` in the example is executed the CPU observes that the object lies in the nursery and the `aputfield_quick` instruction therefore does not violate the write barrier.

In subfigure B) the object reference on the bottom of the stack points to a handle that points to an object in the old generation. When the CPU executes this `aputfield_quick` instruction it will notice that the address of the object is greater than the value in the JBCR register and jump to the write barrier trap handler.

4.2 Modifications in the AVM

The AVM was fully functional at the project start and was capable of handling a garbage collection algorithm. To be able to make the MC^2 implementation efficient some modifications needed to be done. The modifications done to the AVM was:

- Make the array and object structure similar.
- One common function for allocating objects and handles.
- Fixing the mark-sweep implementation.

These modification is described in greater detail in the following subsections.

4.2.1 Make the array and object structure similar

The object and array structure shown in figure 4.1 on page 36 shows that the object and array structure is not equal. The Java Virtual Machine specification [LY99] states that arrays are also objects (A subclass of the class `java.lang.Object`). The headers should therefore be equal to be able to treat arrays as objects. This will also reduce the algorithmic complexity of the garbage collector, because arrays and objects can be treated in a similar manner.

Because arrays also are classes they must have a class data structure that defines every array class and a reference to a lock object in the array header. Before the project start arrays just had a string that defined them and no lock reference.

When moving objects around one need to update the address pointing to the object in the object's handle. It is therefore important that locating a specific object's handle goes as fast as possible. Earlier this was done by traversing a list, but when copying several objects this approach becomes too resource demanding and must therefore be altered.

Implemented changes

The object and array header was made equal by rearranging the fields in the header and by including a lock object for arrays (See figure 4.11). In addition to this a reference to the handle was included in the object and array header to speed up the header location. A description of the items found in the object and array header is found in table 4.5.

The class structures for array classes will be generated dynamically when needed to save space.

4.2.2 One common function for allocating objects and handles

At the project start handles and objects were created separately by calling a function called `jmalloc()` and `halloc()`, for object and handle allocation

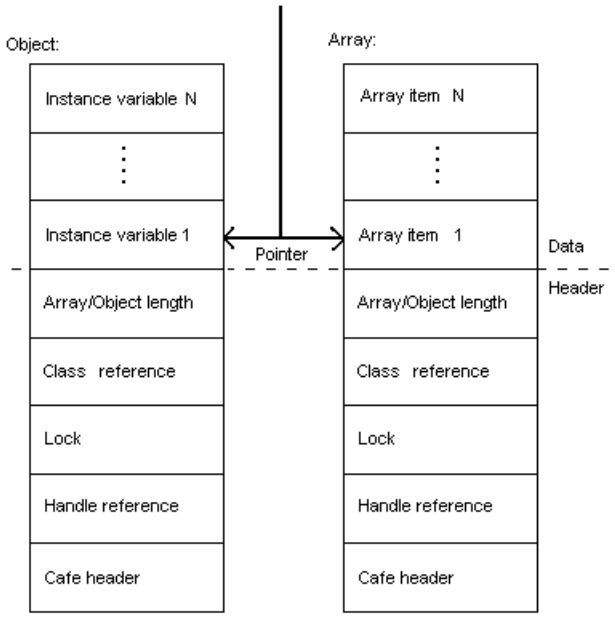


Figure 4.11: The new object and array structure.

Field	Description
Cafe header	The `0xCAFE` word marking the beginning of an object.
Handle reference	A reference to the object's handle.
Lock	A reference to a lock data structure.
Class reference	A pointer to the class structure of the object.
Length	The number of elements in an array or the number of fields in an object.

Table 4.5: The new object and array header items.

respectively. This created a situation where one could allocate an object and not a handle for this object and vice versa. The creation of the object header was left to the function that called `jmalloc()` and `halloc()`. This created the possibility of a situation where the header has not been initialized or a handle not located for the object.

To make object creation more efficient and secure a common code for allocating objects and handles should be made. This code should return a handle when the handle bit is enabled and an object reference when the handle bit is disabled.

Implemented changes

A common assembly macro was created that allocates an object, makes an object header and returns a handle or an object reference (Depending on the handle bit.). The reason for using an assembly macro instead of a C procedure

is that a function call would have reduced the performance of the trap library significantly. For C functions, the macro is put inside an assembly function. This assembly function can be called directly from the C code.

When running with garbage collection, the garbage collector's allocation code is invoked and allocates memory and a handle for the object.

The fusion of the handle allocation and object allocation code made it necessary to change the garbage collection interface. The garbage collector interface is an interface in C that every garbage collector must implement. This makes it easy to make a new garbage collector if necessary. The interface makes it possible for the linker to change the garbage collection algorithm at compile time.

This interface now includes the following functions:

void avm_initializeHeap() Sets up the heap according to the garbage collector's organization of the heap.

void avm_collectGarbage() Invokes the garbage collector.

void *avm_gcObjectAlloc(Klass *klass, int size, int length, int mask) Allocates a new object of size `size` and class `klass`. The `length` field tells the number of fields in the object (if other than in the class data structure.). A pointer to a handle with the mask bits from `mask` is returned to the caller.

int avm_freeMemory() Calculates the amount of free memory and returns it to the caller.

int avm_totalMemory() Calculates the amount of total memory and returns it to the caller.

4.2.3 Fixing the mark-sweep implementation

The rearrangement of the object structure (See section 4.2.1), the rearrangement of the garbage collector interface (See previous section) and the implementation of threads made it necessary to fix the mark-sweep implementation, before benchmarking the mark-sweep algorithm against the MC^2 algorithm.

Implemented changes

The mark-sweep collector was modified to use a C structure for Java object headers. A C structure for the free heap space was also implemented. These changes made it necessary to entirely reimplement the Deutch-Schorr-Waite marker and the sweep function.

The collector was also modified to implement the garbage collector interface. This was accomplished by removing the `avm_halloc()` function, that allocated an object handle, and including its code in the object allocating function (`avm_gcObjectAlloc()`).

The change of the object and array structure also made it necessary to reimplement the `avm_verifyHeap()` function used to verify the heap before and

after garbage collecting to debug the algorithm. This function is very similar to the one found in section 6.2.2.

4.3 Preparation summary

This chapter has described the hardware support necessary to make the MC^2 algorithm work on the Atmel microcontroller and the software changes made in the AVM.

After these changes the AVM is now ready to run the MC^2 algorithm efficiently. The mark-sweep implementation is also fixed and ready to be benchmarked against the MC^2 algorithm.

The design and implementation details of the MC^2 algorithm are presented in the next chapter.

Chapter 5

Designing and implementing the Memory-Constrained Copying garbage collector

This chapter covers the implementation of the MC^2 garbage collector in the AVM. Since the algorithm have been described in section 3.3 only design choices and the implemented solutions are described here.

5.1 Heap layout

The MC^2 heap is partitioned in several windows, as shown previously in figure 3.4 on page 23. The window partitioning creates a few design issues that needs to be solved:

- Data structure to manage the heap windows.
- Managing object handles.
- Handling objects larger than one window.
- Immortal objects.
- Heap management.
- Initial number of windows.

5.1.1 Data structure to manage the heap windows

To manage the heap windows a data structure needs to be created to keep a record of the type, size and free space in each window. This also makes it easier to add or remove fields later if necessary.

Implemented solution

Each heap window will contain a header that contains the items in table .

Field	Description
size	The size of the window, in bytes.
free	The number of free bytes in this window.
type	The type of the window, e.g. old generation window.
current	A pointer to the next field to place data in this window. This pointer points to a place inside the data array described below. The current pointer can be used as a stack pointer in each window, telling the memory system where to put newly created objects.
occupancy	This value displays the number of bytes that marked objects occupy in this window. If the occupancy of a window is more than 95% during copying, the contents of this window will not be copied, due to the small gain in free space and the large overhead.
next	A pointer to another window of the same type. This is used to link together e.g. old generation windows.
data	An array of the data in this window. This is where objects and handles are stored.

Table 5.1: The heap window data structure.

Each of the elements in the header occupies 4 bytes, so the header totally occupies 24 bytes of memory (not counting the data array).

A function for allocating windows was also implemented. This function sets up the window header and allocates enough room for the window on the heap.

5.1.2 Managing object handles

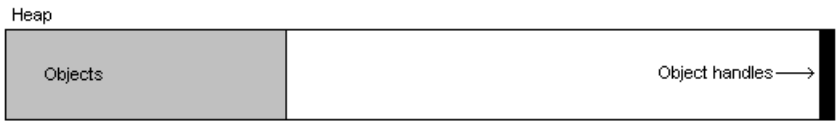


Figure 5.1: Objects and handles on the heap without garbage collection.

In the AVM object handles are, without garbage collection, stored from the bottom of the heap, while the objects are stored from the top. This is shown in the figure 5.1. This arrangement is not suitable for the MC^2 garbage collector because of the window partitioning.

When an object handle is created it cannot be moved. If one should move an object handle the whole heap has to be scanned for pointers to this handle.

This creates too much overhead to be realized in practise. Therefore the object handles must be placed where they can stay for the lifetime of the object.

Because object lifetimes differs, there will be free handles among the occupied handles after garbage collecting. These free handles must be organized in order to reuse the space.

Implemented solution

The MC^2 garbage collector will store its handles in special handle windows on the heap. Initially one handle window will be allocated, but more windows can be allocated when needed.

Because free handles exist among the occupied ones, the free handles are arranged in a linked list to ensure that every handle is reused when deallocated. The current pointer in the handle window will be used to point to the first element of this linked list. The list will be terminated with the word `0xFFFF FFFF`. Figure 5.2 shows an example of this list structure.

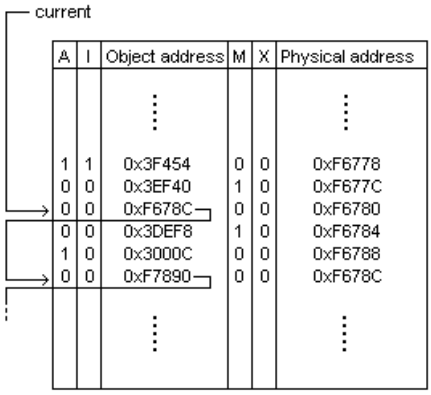


Figure 5.2: Handle free list example.

When an object is deallocated its handle is then linked into the list of free handles. This makes the handle window fragmented, but because every handle occupies 4 bytes of memory every fragment in the handle window can be reused without compacting. A handle window that is completely empty can be deallocated.

5.1.3 Handling objects larger than one window

When the heap consists of windows of a certain size there must be a way to handle objects and arrays larger than the size of one window. If such a method do not exist the Java Virtual Machine would have to exit when trying to allocate an object larger than one window.

In the MC^2 test implementation, by Sachindran, Moss and Berger, all objects larger than 8 kB are stored in a special region for large objects. If an object is larger than 8 kB its size is rounded up to a number of pages and placed in this

region. The rest of the free space in the window are used as space for smaller objects [SMB04].

Implemented solution

The MC^2 solution was implemented and the free space in each window after placing the large object is used as if it was a normal old generation window. The large object is automatically promoted to the old generation.

When the large object is collected the window is emptied and deallocated. Otherwise, if the large object is alive all other objects are copied out of this window and current pointer is set to point right after the end of the large object. The window can now store more old generation objects.

5.1.4 Immortal objects

Immortal objects are special Java objects that never should be garbage collected, even though there does not exist a Java pointer to these from other objects. Such objects are created by the class loader while dynamically loading a class, for example.

Because these objects never will be collected it is more convenient to place these objects in a separate section.

Implemented solution

To have a separate place to store immortal objects, a special heap window for immortal objects will be created. When an immortal object is allocated it will be placed directly in this window and is kept there until the AVM exits.

If the window for immortal objects runs full a new window is created and linked into the list of immortal windows with the `next` pointer in the window data structure.

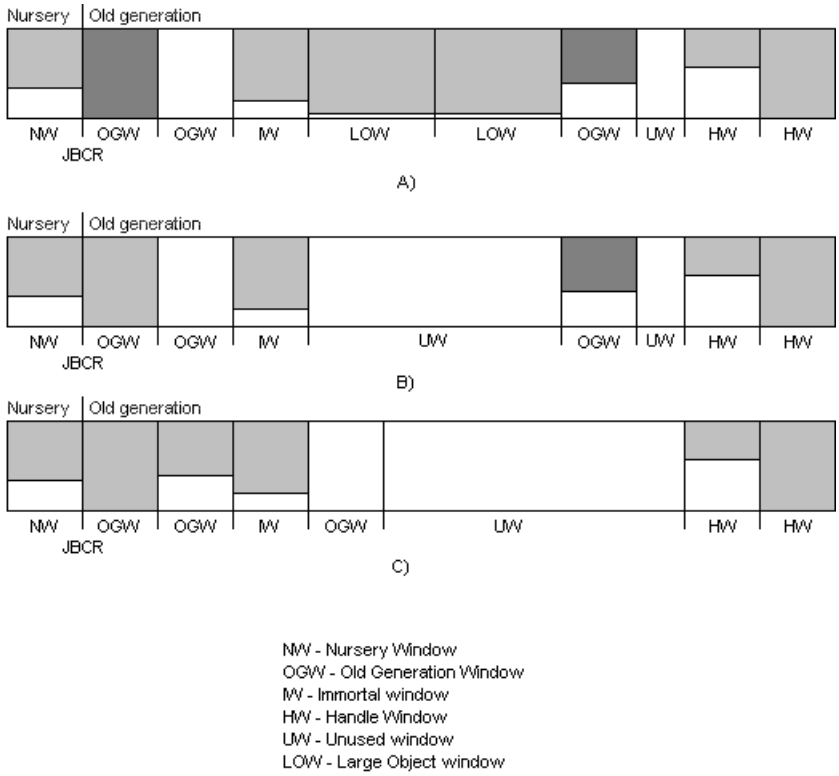
5.1.5 Heap management

Because windows can be allocated and deallocated (like large object windows) a structure for managing the heap must be kept to ensure that no space is lost when allocating.

Implemented solution

To manage the heap a new window type is made, the unused window. This window will contain the space not used by other windows. When allocating a window a suitable unused window is found and the window is divided into the window to allocate and a new unused window with the remaining free space. The unused heap windows will be linked together by the `next` field in the heap window data structure.

If the heap becomes very fragmented, in the old generation copying phase, windows could be moved from one location to another to defragment the heap. Because every live object in a window is copied in this phase there is little extra overhead by performing this operation. Handle windows on the other side cannot be moved until they are empty. If such a window is mixed in between other windows defragmentation may become impossible. Therefore all handle windows are placed from the other end of the heap to reduce the amount of interference with other windows.



Allocated memory is colored grey, marked objects are colored dark grey and unallocated memory is white.

Figure 5.3: Heap defragmentation example.

In figure 5.3 an example heap defragmentation is shown. In subfigure A) the copying phase has just started in the old generation. To simplify nursery collection is left out of this example. All windows have been marked and the two large object windows contain only unmarked objects.

In subfigure B) the two large object windows have been collected and converted to a unused window. This creates a hole in the heap between the immortal window and the old generation window.

In subfigure C) the data in the old generation window to the right in subfigure B) has been copied to the free old generation window. To defragment the heap the old generation window was moved next to the immortal window reducing the number of unused window fragments.

5.1.6 Initial number and size of windows

When the AVM initializes, it will call a function in the garbage collector interface called `avm_initializeHeap()`. This function must set up the heap according to the heap structure for the current garbage collection algorithm.

The MC^2 garbage collector must make one nursery window, one window for immortal objects, one window for handles and some old generation windows. The size of these windows must also be decided.

Implemented solution

The window size will not be fixed to a certain size, but defined in the config file for the AVM. The user can therefore change the size of windows to suit his/her needs.

The initial number of old generation windows will also be user configurable in the same manner.

During the implementation a window size of 8 kB and 10 old generation windows will be used, to stress both the nursery and old generation collection.

5.1.7 Heap windows summary

A short summary of the window types, numbers and size are included here in table 5.2.

Window type	Initial number	Size
Nursery window	1	Defined in config file.
Old generation window	10	Defined in config file.
Handle window	1	Defined in config file.
Immortal object window	1	Defined in config file.
Large objects window	0	N * 4kB blocks (Must be larger than large object.).
Unused window	1	Remaining heap space.

Table 5.2: Summary of window types.

5.2 Write barrier trap

During Java program execution the write barrier will detect pointer mutations in marked objects and the storing of pointers into the nursery. This information must somehow be stored until the garbage collector is invoked again.

It is also important that this process is as fast as possible. These trapped instructions are without the MC^2 garbage collector executed directly in hardware. It is preferable that the write barrier trap therefore does as little as possible.

Implemented solution

The storing of mutated pointers in already marked objects can only happen during the old generation marking phase. During the old generation marking phase a mark stack is present (see section 5.6) and the write barrier can place the mutated pointer directly on this stack. This will ensure that the garbage collector scans this object the next time it is invoked.

Nursery objects that are pointed to from the old generation are given a nursery mark. The remaining unused status bit in the object handle is used for this purpose. The object handle with all status bits are shown in figure 5.4. When the write barrier is triggered because of the storing of a pointer to an object in the nursery the nursery object is given the nursery mark by the barrier.

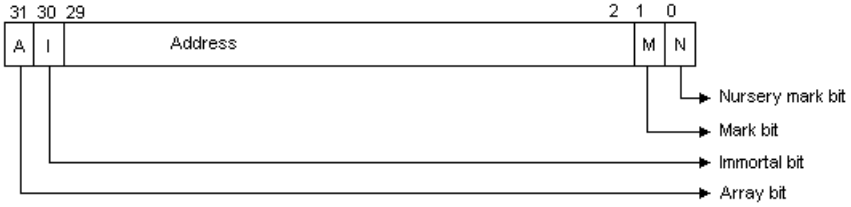


Figure 5.4: Object handle with array, immortal, mark and nursery mark bit.

5.3 Locating object pointers

When scanning the frame stack or an object there must be a way to tell what is a pointer to an object and what is not. In the AVM there is no way of telling during runtime for example what type of value is stored at the top of the stack. This stack lies in the microcontroller hardware and does not contain any type information.

The garbage collection algorithm must therefore treat any value as a possible pointer and is therefore a conservative garbage collector. A method must be made that checks if a value really is a pointer.

Implemented solution

A function was made that checks if a value points to a valid object handle or not. If all the following checks succeeds the value *can* point to a valid object and must therefore be treated as an object pointer:

- The value points into a handle window.
- The value is pointing to a word boundary. Handles are always stored as words along the word boundaries.
- The possible pointer points to a place outside the handle window. This means that it is not part of the handle free list, because a member of the handle free list will always point into the handle window.

5.4 Object allocation

When an object needs to be created the object allocation code of the garbage collector will be invoked. This procedure will allocate room for the object in either the nursery window, immortal window or in a window for large objects.

5.4.1 Allocating memory from the native library

From the native Java methods it may be necessary to allocate objects on the Java heap. This can lead to errors when allocating more than one object:

For instance when allocating a Java string a character array is first allocated and then a string object. When the string object is created the character array is stored in the string object. If the nursery was full after allocating the array, the array would be collected when the string object was being allocated because no reference to the array could be found on the heap. The pointer to the now collected character array would be stored in the string object, creating a dangling pointer.

Implemented solution

Every object allocated on the heap in the native library is temporarily made immortal. After all object pointers have been saved on the frame stack the objects are made mortal again by a recursive procedure.

5.4.2 Locating nursery handles

When performing nursery collection it is important that locating objects handles in the nursery is as fast as possible to discover the objects that have been given the nursery mark.

Another problem is deallocating handles after nursery collection. When the nursery collection is over there are still handles in the handle windows that points to the objects in the nursery. These handles must be found and inserted into a list of free handles to reuse this space.

This information already exists in the list of handles, but scanning through this list, searching for nursery pointers is a too resource demanding job. The reason for this is that there are many more handles pointing into the old generation than there are nursery pointers.

One other solution is to traverse the list of objects in the nursery window and access the handle from the object header. The problem with this solution appears when the collector is moving from one object to the next. The size of the object (including header) used as an offset to the next object is not constant or very easy to establish. The size field in the header tells only how many fields there are in the object or array. The class reference must then be consulted to find the correct size of each field. This value has to be multiplied with the number of fields and added to the header size.

Implemented solution

To speed up the locating of nursery handles a list of pointers to the handles are placed at the other end of the nursery window, growing the other way. This is shown in figure 5.5. This way locating the handles is a very easy task and requires a minimum of resources. This comes at the cost of less space for objects in the nursery.

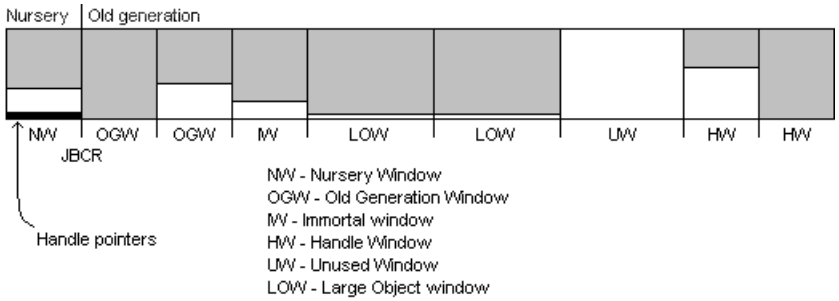


Figure 5.5: Heap example with nursery handle pointers.

In the proceeding "Object-Oriented Architectural Support for a Java Processor" by Vijaykrishnan, Ranganathan, and Gadekarla [VRG98] an average object size is calculated by running a series of Java benchmarks. They found that the average object size is 30 bytes (or around 8 words). When including the object header the average object size becomes 13 words.

This means that in the nursery every 14th word of memory is lost due to the handle pointer and the available space for objects in the nursery is 7% smaller.

With around ten old generation windows, a nursery window, a handle window and a window for immortal objects, the total waste of heap space is lower than 0,5%. In this setting such a small waste is insignificant.

5.5 Nursery collection

The MC^2 paper [SMB04] does not mention the garbage collection method used to copy objects from the nursery to the old generation. One is therefore free to choose a garbage collection method for this window.

In the project leading to this diploma [Amu04] the copying algorithm and the mark-copy algorithm was examined. These two algorithms detects live objects in a region and copies them into another.

The mark-copy algorithm is a modification of the mark-sweep algorithm where the sweep phase has been replaced by a copy phase. The copy phase copies all live objects from the current region of the heap into another region. This requires the region to be scanned twice by the collector, once for mark and once for copy. The nursery collection is not incremental and can therefore use the Deutch-Schorr-Waite algorithm for the marking phase (this method does not require a mark stack).

The copying algorithm only scans the live objects in the nursery region once. It uses the already copied objects in the old generation windows as a mark stack and keeps track of which objects are already scanned with two pointers. When an object is encountered in the mark stack all objects referred to by this object is pushed on the stack. This process continues until there are no more items on the mark stack.

Implemented solution

The copying algorithm looks very promising. It should be faster than the mark-copy algorithm because it only scans the live objects once. This algorithm cannot tell, before starting to copy objects, how much space it will require in the old generation for the live objects. If the old generation runs full during copying it is not in a consistent state and an old generation collection is impossible. An old generation copying in the middle of the nursery collection would also have ruined the mark stack of the algorithm because it is depending on the sequence of objects.

With the mark-copy algorithm the amount of old generation space needed is known after marking all live objects. The nursery collector can then check that there is enough room on the heap and call the old generation collector if necessary. Therefore the mark-copy collector will be implemented as the nursery collector.

The Deutsch-Schorr-Waite algorithm like described in section 3.2 is implemented as the marking algorithm. This algorithm will only scan objects in the nursery and will not follow pointers to objects in other windows.

After every live object in the nursery have been marked all handles that still points to objects in the nursery must be collected and placed in the handle free list.

An average of the percentage of nursery object that have survived, the Nursery Survival rate (NSR), is also kept. This value is used by the old generation marker.

If the old generation collector is in the marking state, the objects copied to the old generation windows are marked with the normal mark bit. Pointers to objects in the old generation that has not been marked are pushed onto the mark stack if the old generation collector is in the marking state.

5.6 Old generation marking

When the old generation windows are 80% full the marking of the old generation starts. The garbage collector will then mark a certain amount of bytes between each nursery collection.

To calculate the amount of bytes to mark the MC^2 paper suggests using the following formula after every nursery collection [SMB04]:

$$numMarkIncrements = availSpace/(NSR * nurserySize)$$
$$markIncrementSize = totalBytesToMark/numMarkIncrements$$

$totalBytesToMark = totalBytesToMark - markIncrementSize$

`NSR` is the average Nursery Survival Rate, calculated by the nursery collector every time it is invoked. The `availspace` variable is the available space in the old generation when old generation marking is triggered. `numMarkIncrements` tells how many nursery collections there will be before the heap runs full. The `totalBytesToMark` value is initialized to the size of the windows to mark at the end of every old generation copying, because in the worst case all objects in the old generation are live. The `markIncrementsSize` gives the number of bytes to mark between the nursery collections.

The reason for calculating this value is that marking should be finished just when there is no more than one single empty old generation window left of space in the old generation. The marker has then spread out its marking as much as it can and therefore created as little program interruption as possible.

Implemented solution

The marking phase starts with putting all root pointers, found in static variables and the frame stack, onto the mark stack. For every object pointer on the mark stack the object is scanned and all pointers from this object that is not already marked are placed on the top of the stack. The marker keeps track of the size of the marked objects and returns if the number of marked bytes is greater than `markIncrementSize`.

Marking terminates when the stack is empty. This puts the garbage collector in copying mode.

5.6.1 Mark stack

When marking objects in the old generation a mark stack needs to be kept in order to find out which objects to mark next. This mark stack must be placed somewhere on the heap or memory must be allocated to keep it.

When scanning large object graphs, the mark stack can overflow. This overflow must be detected and dealt with to ensure that no data is lost.

Implemented solution

During the incremental marking the garbage collector will let the normal Java execution proceed as normal until there is only one empty old generation heap window left. The reason for this is that in the copying phase one heap window must be free to have somewhere to put the objects when copying from one window to another. This window will not be used under the old generation marking and can therefore contain the mark stack for the marker.

If an overflow in the mark stack window is encountered a new window will be allocated for this purpose. This window will be deallocated again when empty.

5.7 Old generation copying

When the copying state is started the occupancy of each window has been calculated. This number tells the rate of live objects in the window. The copying starts with grouping these windows into groups that together fills a whole window. All windows with an occupancy over 95% is considered a high occupancy window and is not copied. This lowers the program interruption [SMB04].

In every subsequent nursery collection, the old generation copying "piggybacks" the processing of one old generation group. This frees up one old generation window for the nursery objects to be put in [SMB04].

Implemented solution

Due to the object handles all objects in a window must be scanned to either copy a live object or to free the object handle. If a grouping of windows is performed all these windows have to be scanned entirely before returning from the copying procedure. In this MC^2 implementation it is therefore better to process one window during each copying increment. This will free up one old generation window every increment and give a shorter program interruption.

Object copying is performed by a special assembly function that is designed to copy objects. It uses the `ldm` (load multiple) and `stm` (store multiple) instructions to speed up the copying. These instructions loads and stores multiple words from the main memory and into the register file.

When all live objects from the old generation windows are copied the old generation garbage collector is put back in the normal state.

5.8 User triggered garbage collection

In Java the user can trigger the garbage collection at any time by calling the method `System.gc()`. This invocation can happen at any time in the garbage collection cycle, like in the middle of the old generation marking. The garbage collector must then stop garbage collecting in incremental steps and run the collector until completion.

Implemented solution

When `System.gc()` is called the state of the garbage collector is read. If the garbage collector is in the old generation marking state or the old generation copying state, the old generation collector is invoked and run until the state is back to normal again.

When in the normal state a nursery collection is performed, followed by old generation marking and old generation copying until back to normal state again.

5.9 Remembered sets, sequential store buffers and card tables

The MC^2 article states that to record stored pointers between windows it uses remembered sets, sequential store buffers and card tables to remember these pointers. These data structures makes it easier to perform marking because one can scan the heap linearly instead of scanning it like pointer tree during the marking phase [SMB04].

These data structures must be updated every time a pointer store is performed. This means that `putfield`, `aastore` and `putstatic` should run entirely in software, both performing the actual store and updating these data structures.

In addition these data structures would have to be stored in each heap window, wasting some space on the heap. The MC^2 article reports that this overhead occupies at most 5% of the heap [SMB04].

Making the `putfield`, `aastore` and `putstatic` bytecodes run entirely in software is not preferable, because this would mean that the execution time of these instructions would be severely reduced.

Remembered sets, card tables and sequential store buffers will therefore not be implemented in this MC^2 implementation.

5.10 Source code

The garbage collector's source code can be found on the CD found in appendix A. The code in this appendix is the property of Atmel and is not available for the general public.

5.11 Implementation summary

The Memory-Constrained Copying algorithm is now implemented in the AVM. To verify that it behaves correctly it must be tested functionally. The test setup and test procedures are presented in the next chapter.

Chapter 6

Testing the garbage collector implementations

To test the correctness of the garbage collection implementation, the AVM is run in a sophisticated test environment. This environment is designed to test Atmel's 32-bit microcontroller, because it is not manufactured yet. This chapter includes a description of the test setup, test procedures, test programs and the result of the testing.

6.1 Test setup

Because the microcontroller is not realized in silicon yet, hardware and software simulators are used to test the garbage collector. This section describes general hardware-software co-simulation techniques and the two test environments used to test the algorithm, the software simulator and the hardware simulator.

This introduction was written in the project leading to this diploma [Amu04] and was revised and included here.

6.1.1 Hardware-software co-simulation

Hardware-software co-simulation is used to validate the hardware portion as well as the software run on a system and the interaction between them. In the construction of a complex embedded system a hardware-software co-simulation provides the designers with valuable information about the correctness of the design. It also makes it possible to test different hardware and software designs without taping out a new part every time. This is both time and cost saving [HB97].

The simulation can be done in several detail levels. The more detailed, the slower the simulation will run. This list describes a few detail levels (The number of instructions per second is just a relative number. Assuming software simulators run on the same computer.):

The nanosecond accurate processor model This technique models the hardware in great accuracy, at a transistor level. A standard computer may run 1 to 100 instructions per second with this model.

The cycle accurate processor model This model has a structure very similar to the real processor, with pipelines, interlocks and functional units. Around 50 to 1000 instructions are executed per second.

The instruction set accurate processor model This processor model models the instruction set architecture of the target processor, but does not provide correct timing. Units like pipelines and caches are not included. This model can run about 10000 to 100000 instructions per second.

The model-free synchronizing handshake In the synchronizing handshake the hardware is run in a hardware simulator and synchronized with the software with a special handshake. The software is compiled for the host system. The performance of the hardware simulator limits the execution speed of this model.

The virtual operating system This model abstracts both the processor and operating system away and provides the software with an interface that resembles the operating system's. This provides a very good relative speed, but it hides away all the hardware details.

The bus functional processor model The processor in this model is replaced with a set of test vectors. This helps validating hardware, but does not aid the software validation.

The designers must choose a detail level that provides just enough details and minimizes simulation time.

Table 6.1 shows the corresponding hardware-software co-simulation levels for the software and hardware simulators used in this project.

Co-simulation level	Simulator
The nanosecond accurate processor model	Hardware simulator
The cycle accurate processor model	Software simulator
The instruction set accurate processor model	Instruction set simulator
The model-free synchronizing handshake	
The virtual operating system	
The bus functional processor model	

Table 6.1: The relationship between the simulators and the hardware-software co-simulation levels.

6.1.2 Software simulator

The software simulator is developed at Atmel and is a software model of the microcontroller at a cycle accurate level. See section 6.1.1 for a description of these levels.

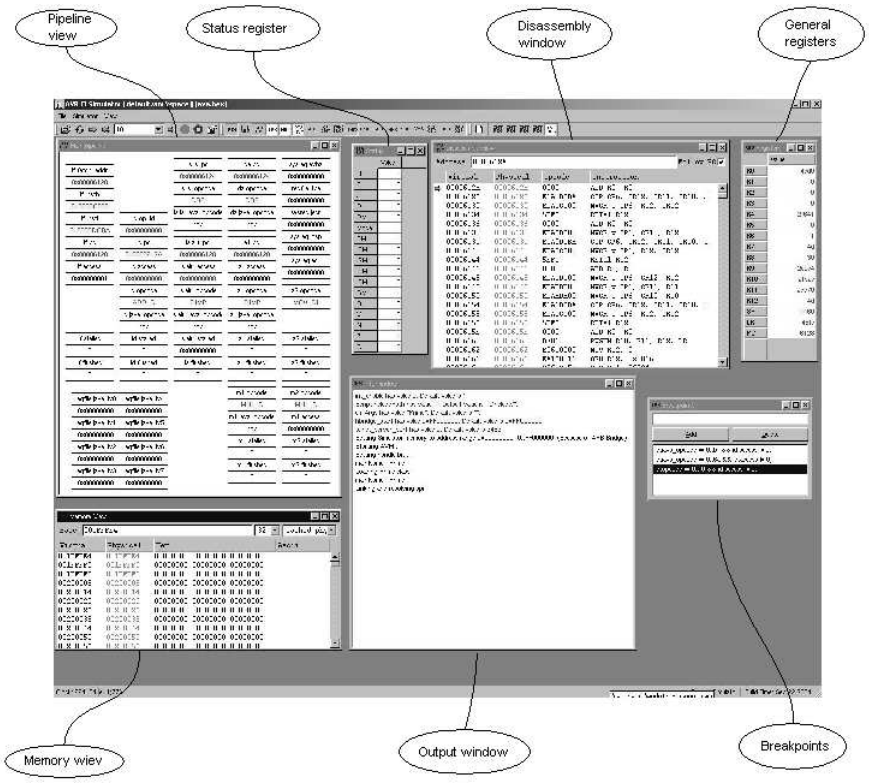


Figure 6.1: Software simulator screen shot.

The microcontroller model is encapsulated in a Graphical User Interface (GUI). A picture of this user interface is shown in figure 6.1. The simulator is capable of showing a lot of information in various windows. A short description of some of these windows follows here:

Breakpoints This window allows the user to set various breakpoints. The breakpoints are defined in a C like syntax, like "`(R6 & 0x00405000) && R3 > 0;`".

Memory view The memory view shows a map of the whole address space of the microcontroller, with both virtual and physical addresses and the values stored here.

Pipeline view The pipeline registers are shown in this window. This is a very useful tool for debugging complex instructions.

Status register The status register is shown in this window. The status register shows if the microcontroller is in Java mode and if the handle bit is set, among other things.

General registers These registers represents the top of the Java stack in Java mode.

Disassembly window The disassembly view is capable of showing the disassembled program code in both assembly and C code when available.

Output window The output window displays the output of the program, allowing the program to communicate directly with the user.

The software simulator is a very useful tool for complex debugging operations. Because the microcontroller model is rather complex the simulation takes quite a while. On a reasonably fast computer it takes about 6 minutes for the AVM to finish classloading and starting to run its first Java instruction. This is without classfile verification, which adds around 15 minutes to the execution.

An instruction set accurate simulator plug-in exists to this simulator. This plug-in removes all the complexity of the pipeline and caches and the simulation therefore runs about 1000 times faster.

6.1.3 Hardware simulator

A nanosecond correct simulator (see section 6.1.1) is also developed for this microcontroller and runs on special hardware. Figure 6.2 shows how the components are connected and how they communicate. This hardware simulator runs up to 1000 times faster than the cycle accurate simulator.

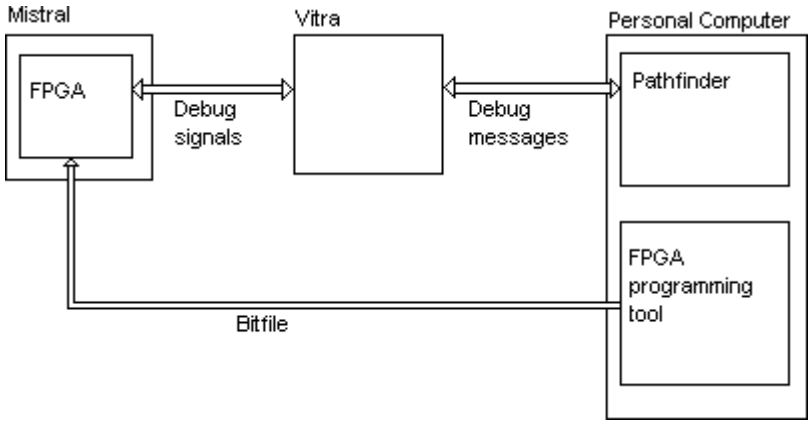


Figure 6.2: The hardware test setup.

Mistral

The Mistral is a circuit board with a large Xilinx FPGA (Field Programmable Gate Array). This circuit board was developed by Atmel Rousset, France. Connected to this card is a top card, which contains many connectors for I/O devices like a LCD screen, PS/2 (for mouse and keyboard), DDR RAM, PCI card, audio and VGA to name a few. The Top card was developed by Atmel Norway.

Vitra and Pathfinder

The Vitra and Pathfinder are parts of a debug system for embedded processors developed by Ashling. The Vitra communicates with the microcontroller run-

ning on the FPGA via the NEXUS 5001 protocol (also known as IEEE-ISTO 5001™-2003).

The Vitra is connected to a personal computer via ethernet or USB. The Pathfinder debug software can then communicate with the Vitra and display the state of the microcontroller in its Graphical User Interface. For more information, see Ashling's Pathfinder home page [Ash05].

6.2 Testing procedures

A brief summary of the programs used to test the algorithms is presented in this section, as well as the functional test specification and the test method.

6.2.1 Test programs

This section describes the Java test programs used to test the garbage collection implementations.

The Prime and GCTest programs were initially written for the mark-sweep algorithm implemented in the project leading to this diploma [Amu04]. They have been rewritten and made more suitable for testing the MC^2 implementation.

Prime

This test program calculates the first 100 prime numbers and prints them out on the standard output. The algorithm was found in the book "Programming languages, concepts and construct" by Ravi Sethi [Set89]. Prime creates an object for each prime number and additional objects every time it prints out a prime number and produces a rather large call stack.

The program also have stored a static array of the primes that is used to verify that every prime number is correct.

This program terminates very quickly, but during its execution it creates some objects and arrays and it is therefore suitable to test the garbage collection and object allocation code and verify that this behaves correctly.

InfinitePrime

This program calls the Prime test program in an infinite while loop. The program will create a lot of objects very rapidly, stressing the nursery collection and old generation marking and copying procedures.

GCTest

The GCTest is a program designed to stress test the garbage collection algorithm. It calls a number of small programs randomly.

This program tests all aspects of the garbage collector. The small programs creates a lot of different objects and arrays, from single stand alone objects to large binary trees.

Below follows a short description of each test program:

Prime The prime number program presented above.

BioMaxima This test program was written to test that the algorithm functioned with a large number of objects on the heap. The test program uses a biologically inspired algorithm to find the maxima of a complex function with a large definition space. This test program was inspired by Dr. ing. Gunnar Tufte's lecture about biologically inspired hardware systems in the course "TDT 1 Advanced Computer System Design".

This algorithm first makes an initial generation of 32 random solutions. All these solutions are run through a complex function and their scores are recorded. The list of solutions in the generation is then sorted on their score. The top five solutions are transferred to the next generation. The remaining solutions are generated by mutating and combining random solutions from the previous generation. When the maxima is found, the number of generations is displayed and the algorithm terminates.

The algorithm is designed to use a lot of memory to stress the garbage collector's object creation code.

TestNative A small program that tests the native functions of the AVM. This includes reading from a file, opening a file, copying an array and dynamically loading a class.

ArrayNewTest This program makes a few arrays of different types (short, integer, long), some multi-dimensional arrays and a few objects.

This program ensures that the array classes that are not used very often in normal programs (like arrays of doubles for instance) works.

GCBench The GCBench algorithm is a Java garbage collection benchmark algorithm written by John Ellis and Pete Kovac of Post Communications [EK05]. It creates a long lived array of doubles and many long-lived and short-lived binary trees.

This algorithm stresses the nursery collector, the old generation marker and the old generation copying code, because a tremendous amount of objects are created to build all the binary trees.

6.2.2 Functional testing

To test that the garbage collector and memory allocator functions properly a function, `avm_verifyHeap()`, has been implemented in the garbage collector. This function scans the heap before and after each nursery and old generation collection to verify that the heap is in a stable state both before and after collecting garbage.

The first thing this function checks is that the heap window data structure is intact. The following checks are made:

- Every heap window is reachable by the program code (by a direct pointer or through a linked list) and has the right type.
- The size field in each window is correct and that a new window follows after this window.

The `avm_verifyHeap()` function then scans all heap windows except handle windows. During this pass the function checks the following in each window:

- Every object and array is connected to a valid handle and that this handle points back to the object again.
- Every object and array has the `0xCAFE` header intact.
- The size of each object and array equals a number of whole words (4 bytes) and that an object follows directly behind this one.
- The free space in each window equals the size of the window minus the size of every object in it.

Then the handle windows of the heap is scanned and the following checks are made in every window:

- Every handle points to a valid object or array and that the handle reference in the object header points back to the handle again.
- Every free list item points to a another free list item.
- Every word scanned is either a valid handle or in the free handles list.
- The handle free list terminates with the word `0xFFFF FFFF`

The `avm_verifyHeap()` function can be turned off at compile time, by the definition `VERIFY_HEAP` in the AVM configuration file. This makes it possible to use the verifier as a debug tool, if needed after this project has ended.

If an error is encountered in a window the function will report which error occurred and the window it occurred in. It will also print a map of the heap windows to ease debugging.

This function can not test that the live object graph is the same before and after garbage collecting. This would in addition to the tests performed be an ultimate test of the garbage collector's correctness. Such a function will require a lot of storage and processing power to perform. It is also almost impossible because of the lack of type information in stored values.

During algorithm testing and development, this function will be executed every time before and after the garbage collector is invoked. By running the verifier every time most functional errors will be discovered and corrected at an early stage.

6.2.3 Test method

To make the testing of the garbage collector easier the test programs run on the AVM will be of increasing complexity. This would hopefully lead to the discovery of bugs early in the test procedure, because the less complex programs are easier to debug. The test method can be seen in figure 6.3.

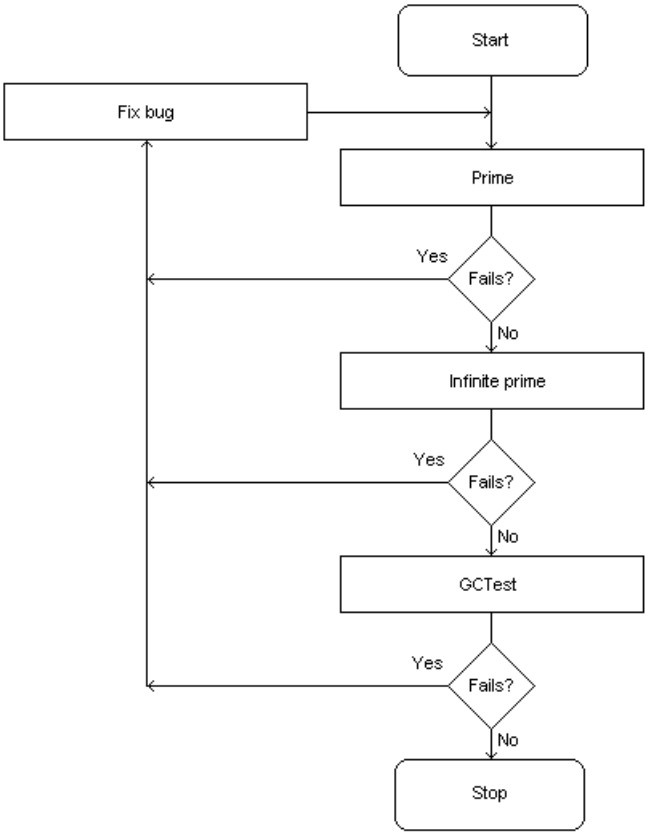


Figure 6.3: Flow diagram of the garbage collector test method.

To test the nursery collector and heap initializing function the Prime test program will be run first. The second test program is the Infinite Prime program which stresses the nursery and old generation collector. The last test program is the GCTest program. This program is much more dynamic than the other two and tests all aspects of the garbage collector and creates a rather complex object graph.

If an error is found in the garbage collector, it is fixed and the testing starts over again with Prime.

6.3 Testing results

The `avm_verifyHeap()` function was invoked every time before and after the garbage collector was called. When this function reported an error it was immediately fixed and the test case rerun to ensure that the bug was corrected. The test have been run on both the MC^2 algorithm and the mark-sweep algorithm.

This function have been very useful and have led to the discovery of many bugs that would have been hard to locate otherwise. A description of the most critical errors caught with the `avmverifyHeap()` function can be found below:

- An addition error made the size of free heap windows four bytes less than it should be. Over time this meant that the heap kept getting smaller and smaller until a new window could not be allocated.
- An error in the old generation copying phase that did not insert a large object window in the right list after copying out every live object, except the large object itself.
- A falsely deallocation of handles of live objects in certain situations. The object itself was copied and the handle updated, but after copying the object the handle was deallocated.

6.4 Test summary

The MC^2 and mark-sweep implementation have been put thorough a thorough test and it has been verified that the heap structures and objects were in a sane condition both before and after garbage collection.

The garbage collectors are after the functional testing more suited to run the benchmarks with fewer problems than if the testing not had been performed.

First a benchmark must be chosen to test the garbage collection algorithms with. The selection of a benchmark is following in the next chapter.

Chapter 7

Choosing a garbage collection benchmark

In this chapter proper benchmarks for the comparison of the mark-sweep and Memory-Constrained Copying implementations are chosen. First an introduction to benchmarks and benchmarking is included. Then a few factors that influence the benchmarking is discussed before the available benchmark candidates are discussed and a benchmark for comparing the garbage collector's performance is found.

7.1 Benchmarking and benchmarks

The information in this section is gathered from Patterson and Hennessy's book "Computer Architecture: A Quantitative Approach" [PH90].

To measure the relative performance between two entities (like computers or algorithms) it is important to have a set of tests that can be run on both these entities to produce some kind of score that is comparable. A set of tests like this is often referred to as a benchmark.

Benchmark programs span from real programs like Microsoft Word to programs tailor made to test a computer's performance, like Dhrystone (A classical synthetic benchmark [Wei84]). According to [PH90] the most ideal way to test the performance of a computer is:

> *"A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. To evaluate a new system the user would simply compare the execution time of her workload – the mixture of programs and operating system commands that users run on a machine [PH90]."*

This test is very hard to perform in practise as it is a non-autonomous test that require a real person. Therefore benchmarks are made that try to simulate

the users stimuli on a system and by that create a measure of computer's performance. Benchmarks are classified by the level of accuracy which they simulate the real world applications.

7.1.1 Benchmark categories

In [PH90] benchmarks are categorized in five different levels. Below is a description of these levels in decreasing order of accuracy and prediction:

Real applications

Real applications are off-the-shelf programs that are run on the target system. These applications are sometimes less suitable as benchmarks as they often are dependent on the operating system or compiler. To port these programs to another platform one probably have to modify the source and maybe eliminate an important activity like graphical user interface.

Modified or scripted applications

These applications use the building blocks of off-the-shelf applications to build a benchmark. These building blocks are either modified or scripted with a simulation of user stimuli. This can both enhance portability or make the benchmark focus on one specific system unit to test, like CPU throughput.

Kernels

Kernels are extracts of small key pieces of program code from off-the-shelf programs used to evaluate performance. Kernels are best used to isolate performance of individual features of a machine to explain the differences in performance of real programs.

Toy benchmarks

Toy benchmarks are often between 10 and 100 lines of code and produce a result the user already know before running the program. Examples of such programs are the tower of Hanoi or quicksort. These benchmarks are easy to write and will run on almost all platforms.

Synthetic benchmarks

Synthetic benchmarks try to match the average frequency of operations and operands of real world programs. Synthetic benchmarks lies even further from reality than kernels because kernel code is extracted from real programs, while synthetic code is created artificially to match an average execution profile.

7.1.2 Benchmark suites

Benchmarks are often delivered as a benchmark suite, consisting of many different benchmarks. These suites try to measure the performance using a variety of benchmarks so that the weakness of one benchmark is lessened by the presence of the others.

Embedded benchmark suites

Embedded benchmark suites are harder to define due to the large range of embedded system requirements (hard realtime, soft realtime and overall cost-performance). Many designers make embedded benchmark suites that reflects the application performance as a kernel or as a modified version of the application.

For embedded systems the most recognized benchmarks are the EDN Embedded Microprocessor Benchmark Consortium's (EEMBC, pronounced "embassy") benchmark suite . The EEMBC benchmark suite consists of 34 benchmarks from five different industries: automotive/industrial, consumer, computer networking, office automation and telecommunication [PH90].

7.2 What to measure

In chapter 3 the mark-sweep and MC^2 algorithms were compared against each other, theoretically, in four different categories: program interruption, throughput, algorithmic complexity and memory utilization.

In the following practical comparison it would be nice to be able to test the implementations in all these categories, but the resources in this project does not allow this. Only the program interruption and throughput will be measured, as stated in the project assignment (see chapter 1).

7.2.1 Program interruption

Program interruption is the time the user's Java program is interrupted from its normal behavior so that the garbage collector can do its work. It is therefore the time from the garbage collector algorithm is invoked and until it returns back to the Java program again.

In the book "Usability Engineering" Jacob Nielsen [Nie93] mentions the three important limits when measuring program interruption (or response time as it is called in his book). These limits are described in table 7.1.

The goal for each garbage collection implementation is therefore to get the program interruption down to under 0.1 seconds or less every time the garbage collector is invoked.

Note that it is only the actual garbage collection that is timed during the benchmarking. Object allocation is not considered garbage collection and is therefore excluded.

Limit	Description
0.1 second	is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.
1.0 second	is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.
10 seconds	is about the limit for keeping the user's attention focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, because users will then not know what to expect.

This table is quoted from [Nie93].

Table 7.1: Program interruption limits.

7.2.2 Throughput

The throughput of the garbage collection implementation says how fast the garbage collector can collect each garbage object. Or said in another way how much overhead the garbage collection implementation puts on the normal Java execution. The lower throughput, the longer total execution time the user program will have.

The goal for the garbage collection implementation is to have as high throughput as possible to create as little overhead as possible for the Java program.

7.2.3 Factors that influence the measuring

The goal of the benchmarking is to test the relative performance of the garbage collection algorithms, but when running the tests it is the actual implementation of these algorithms that is tested against each other. The code effectiveness will therefore affect the performance of the algorithms in one way or another.

7.3 Benchmark candidates

The AVM implements the J2ME CLDC 1.1 standard (see section 2.1.1). All programs run on the AVM must therefore be compatible with this standard.

Unfortunately many real life programs and benchmarks, like Jbenchmark used to test J2ME CLDC mobile telephones, also require that the MIDP profile is implemented in addition to the CLDC configuration.

The MC^2 article reports that they used a benchmark from Standard Performance Evaluation Corporation (SPEC) [SMB04]. This benchmark requires

that the JVM implements the J2SE 1.1 API and is therefore unsuitable for the AVM [SPE05].

A wide search has been performed on the web and at Atmel to find benchmarks for the garbage collector. The alternatives found are presented in table 7.2.

Benchmark	Suitable for the AVM	Requirements
specJVM	No	J2SE API requirement
JBenchmark	No	J2ME MIDP requirement
EEMBC Java benchmarks	Yes	
CaffeineMark	Yes	originally requires J2SE, but an embedded version exists that use J2ME CLDC v 1.1
VolanoMark	No	Server benchmark, requires J2SE
JMark	No	Network computer benchmark, requires J2SE

Table 7.2: Benchmark alternatives.

One more option is also possible: To write a new benchmark for a J2ME CLDC v 1.1 Java Virtual Machine. This option is also included in the evaluation.

The three benchmark alternatives suitable for the AVM are listed below:

- Write a self-composed benchmark.
- The EEMBC Java benchmark suite.
- The CaffeineMark benchmark.

A description of these candidates follows in the subsections below.

7.3.1 Write a self-composed benchmark

A self-composed benchmark could be written that test an JVM implementation with a high level of accuracy. The resources in this project are too few to begin such a task.

A not so resource demanding option is to use some of the programs from the functional testing of the garbage collectors. The test programs, from the GCTest program (See section 6.2.1), could be timed and run to produce some kind of score, telling how fast the execution was.

This benchmark would then fall in under the toy benchmark category.

7.3.2 The EEMBC Java benchmark suite

The EEMBC Java benchmark suite is designed to test the performance of a J2ME implementation of a Java Virtual Machine. A white-paper for this benchmark suite can be found on the EEMBC Java Benchmark web page [EEM03].

The EEMBC Java benchmark suite consists of six benchmarks that uses the key pieces from real life programs to perform the benchmarking. The EEMBC Java benchmark suite therefore falls in under the kernel benchmark category.

A description of these six benchmarks are found below [EEM05]:

PNG image decoder This benchmark decodes a 19 kB PNG (Portable Network Graphics) image. The decoding is mathematically intensive and does a lot of array copying.

Chess game The chess game benchmark plays three games of chess with ten moves against itself. The algorithm builds up a tree of legal moves and uses a heuristic search to find the best move. This method does a lot of method calls, allocates many arrays and is intensive in native code use.

XML parser This benchmark parses an XML document into a Document Object Model (DOM) tree and performs node searching and node manipulation of the tree. This benchmark allocates a lot of objects and performs many string and string buffer manipulations.

Cryptographic package The cryptography benchmark encrypts a 4 kB text string, decrypts it again and compares the two plain text pieces. It uses the following cryptographic algorithms: DES, 3-DES, IDEA, Blowfish and Twofish. This benchmark is mathematically intensive and uses large byte arrays as buffers for the plain- and crypto text.

Regular expression package This benchmark scans a text string and tries to match it with a regular expression. The benchmark exercises String matching and I/O.

Parallel benchmark The parallel benchmark performs merge sorting and matrix multiplication using a varying number of parallel Java threads. This stresses the Thread implementation of the Java Virtual Machine.

7.3.3 The CaffeineMark benchmarks

CaffeineMark was written by Pendragon software in 1997. It tests the different parts of a JVM and produce a geometrically average score that tells how fast the execution was. CaffeineMark can be found on the home page of Pendragon software [Pen05].

The original CaffeineMark benchmark consists of the following parts:

Sieve The Sieve of Eratosthenes. Finds prime numbers.

Loop The loop test uses sorting and sequence generation to measure compiler optimization of loops.

Logic Tests the speed with which the virtual machine executes decision-making instructions.

Method The Method test executes recursive function calls to see how well the JVM handles method calls.

String Various string manipulation tests.

Float Simulates a 3-D rotation of objects around a point.

Graphics Draws random rectangles and lines.

Image Draws a sequence of three graphics repeatedly.

Dialog Writes a set of values into labels and editboxes on a form.

In the embedded CaffeineMark the Dialog, Image and Graphics tests are removed because these functions are not part of the CLDC standard.

The CaffeineMark benchmark does not report its accuracy level on the home page. It is therefore assumed that the benchmark would, by looking at the description of the tests, be in the toy benchmark category.

7.4 Choosing a benchmark

The goal of the benchmarking is to measure the garbage collector's program interruption time and throughput when running Java programs on the microcontroller.

The benchmarking should simulate the operations performed by a real person as closely as possible and therefore have a high level of accuracy. This requirement makes the EEMBC Java benchmark suite the preferred benchmark, because it is a kernel level benchmark, because the self-composed benchmark and CaffeineMark would be in the toy category. There are a things to consider before making the decision final.

CaffeineMark tests various parts of the JVM, but it does not include a test of the memory management system. The only test in CaffeineMark that uses objects are the String test, which allocates a few StringBuffers, and float which allocates some matrixes. This makes the CaffeineMark unsuitable because object creation will be a insignificant part of the score.

Some of the EEMBC Java benchmarks did not work correctly on the AVM at the project start, because of recent changes in the AVM implementation. The debugging of these benchmarks is hard due to their complexity. It was therefore hard to estimate the amount of work needed to make them function properly again.

The self-composed benchmark's test programs already work as they have been used in the GCTest algorithm, but a method for calculating the throughput must be made because some of the tests make the count register (used to measure the program execution time) overflow. A sequence of the test programs must also be established and the benchmark must then be tested before performing the actual benchmarking.

All in all it is more work to make the EEMBC Java benchmark suite function, but the gain in accuracy of the benchmark makes the EEMBC Java benchmark suite the most reasonable choice. The EEMBC Java benchmark suite is therefore selected as the benchmark to use when comparing the garbage collectors.

To make sure that some benchmarking will be performed before the project ends the focus was kept on one benchmark at a time. The most interesting benchmarks, from a garbage collection viewpoint, will be fixed first. The following order was chosen:

1. XML parser.
2. Chess game.
3. Regular expression package.
4. PNG decoder.
5. Cryptographic package.

The parallel benchmark is excluded because the thread support is not fully implemented or tested in the AVM at the time of the benchmarking.

7.5 How to measure program interruption and throughput

This section describes how the program interruption and throughput will be measured.

7.5.1 Program interruption

To measure the program interruption time every call to the garbage collector must be timed and the result must be recorded. `The clock()` function call will be used to time the execution of the garbage collector. This function returns the value of the processor's count register, which is incremented for each clock cycle. To get the execution time one must simply divide the number of clock cycles with the clock frequency of the microcontroller.

In addition to this a C function is written that records the frequency of program interruption times in a table. This table is printed out at the end of the Java program execution. The average and maximum program interruption time is also calculated and displayed.

7.5.2 Throughput

The EEMBC Java Benchmark suite produces a score after each benchmark is finished telling how fast the benchmark was completed relative to a specific Java Virtual Machine run on a specific processor. This score can be used to measure how much overhead the garbage collector adds to the user program. The score is a linear score, proportional to the program execution time, and can therefore be used to compare directly how much faster an implementation is compared to another. Note that the throughput is inverse proportional with the program execution time.

The score of the benchmarks is calculated by invoking the Java native method `system.currentTimeMillis()`. This method used the `clock()` C function that reads a 32-bit count register that tells how many clock cycles the CPU has been executing. Because it is only 32 bits in this counter the counter will overflow after a short time and therefore make the benchmark score negative or at least unreliable.

The `system.currentTimeMillis()` method was therefore reimplemented using special performance counters in the CPU that generated an interrupt when overflowing. An interrupt handler was also written that took care of this incident.

7.6 Choosing benchmark summary

Three benchmark candidates were found for the benchmarking of the garbage collectors, the EEMBC Java benchmark suite, CaffeineMark and writing a self-composed benchmark. The EEMBC Java benchmark suite was selected due to its higher accuracy of predicting the performance of the garbage collectors.

The next chapter contains the results of the benchmarking of the garbage collection algorithms.

Chapter 8

A comparison between the mark-sweep and the MC^2 implementations

This chapter contains the results of the benchmarking of the mark-sweep and the Memory-Constrained Copying implementations. The test system is first described. The test parameters are specified and followed by a description of the test cases. After this the results from the program interruption and throughput tests are presented and commented.

8.1 Test system

The test system is the hardware simulator described in section 6.1.3. The AVM will be run on the Mistral board with 2 MB internal memory and at a clock frequency of 25 MHz. The real microcontroller will run at a speed of 100 - 150MHz, so the speed of the Mistral is four to six times less.

Due to the reduced speed the benchmark results will be multiplied by five to simulate a clock speed of 125 MHz. This will produce a result that is approximately equal to the result expected from the real microcontroller.

8.2 Test parameters

There are a few parameters that are interesting when testing the garbage collector. These parameters and their values in the test cases are discussed in this section. To limit the number of test cases only the two most significant parameters are tested, the heap size and the MC^2 heap window size.

8.2.1 Heap size

The size of the heap is an interesting parameter when testing the garbage collection implementations. The heap size will affect the mark-sweep implementation the most because it always uses the maximum available space on the heap to store objects in. The MC^2 algorithm will use as much space as it needs and keep room on the heap to allocate windows for handles and normal and large objects, and will therefore be less affected.

Because the garbage collector will run in a Java Virtual Machine that implements the CLDC specification (See section 2.1.1.) it is expected to be run in an environment with a small Java heap. The benchmarking will therefore be most realistic with a small heap size.

A CLDC device is designed to run with a Java heap of 32 kB or more [Top02], but the benchmarks are resource demanding and a larger heap size must be used. Ideally it would be preferred to test the garbage collectors with three different heap sizes: small (500 kB), medium (1 MB) and large (4 MB).

Unfortunately, the Atmel test system setup development status at the time of benchmarking did not allow for testing with 4MB of Java heap. The garbage collectors will therefore only be benchmarked with a heap of 500 kB and 1 MB.

8.2.2 Memory-Constrained Copying heap window size

The MC^2 heap window size has an impact on the MC^2 implementation's performance, because it governs the frequency of garbage collection cycles. The smaller the window size is the more often nursery and old generation collections must be run.

Because this parameter only affects the MC^2 implementation it will be run in two versions, one with a small heap window size (8 kB) and one with a large heap window size (16 kB).

8.3 Test cases

This section contains a description of the test cases that will be used in the benchmarking. Every benchmark will be run with the MC^2 implementation with two different window sizes and the mark-sweep implementation in turns. A description of the test cases can be found in table 8.1.

At the time of the benchmarking only three of the five benchmark candidates were working: kXML, chess and regular expression (regexp). The benchmarking will be carried out with only these three and the remaining three will be left as future work.

The kXML benchmark also did not work with a 500 kB heap, so this test is excluded. In both MC^2 test cases and the mark-sweep test case the reason for the failure is fragmentation, on a heap window level and object level respectively.

The mark-sweep heap fragmentation is not possible to counter because mark-sweep cannot defragment the heap. The MC^2 algorithm can defragment the heap, but because of project resource issues this problem is left as future work.

Test case	Benchmark	GC Algorithm	Heap size	MC^2 window size
1	Chess	MC^2	500 kB	8 kB
2	Chess	MC^2	500 kB	16 kB
3	Chess	Mark-sweep	500 kB	N/A
4	Chess	MC^2	1MB	8 kB
5	Chess	MC^2	1MB	16 kB
6	Chess	Mark-sweep	1MB	N/A
7	Regexp	MC^2	500 kB	8 kB
8	Regexp	MC^2	500 kB	16 kB
9	Regexp	Mark-sweep	500 kB	N/A
10	Regexp	MC^2	1MB	8 kB
11	Regexp	MC^2	1MB	16 kB
12	Regexp	Mark-sweep	1MB	N/A
13	kXML	MC^2	1MB	8 kB
14	kXML	MC^2	1MB	16 kB
15	kXML	Mark-sweep	1MB	N/A

Table 8.1: Test cases for garbage collection benchmarking.

8.4 Test results

This section contains the test results from the test cases. Only selected figures from the benchmarking are shown, the rest are displayed in tabular form. The complete data set from the benchmarking can be found in an Excel file on the CD in appendix A.

8.4.1 Program interruption time

Figure 8.1 shows the program interruption time distribution for test case 1, 2 and 3 running the chess benchmark. In the figure one can observe that the program interruption time distribution for the MC^2 implementation lies in the range zero to four milliseconds and that the mark-sweep lies in the 40 to 58 millisecond range. The number of occurrences of MC^2 program interruptions are larger than the number of occurrences with the mark-sweep implementation.

This figure shows that the MC^2 implementation will interrupt the Java programs much more often than the mark-sweep implementation, but the interruption time is much shorter.

A summary of the result from all test cases are shown in table 8.2. This table shows the average and maximum program interruption time as well as the number of garbage collector invocations.

The table shows that the trend from the last figure is repeated in all test cases: MC^2 gives many small program interruptions, but mark-sweep produce a few longer interruptions.

In the kXML benchmark many objects was created in a short period of time and stressed the garbage collectors. This can be observed in the number of garbage collections invocations and the program interruption time. The mark-sweep implementation pauses the Java execution on average for about one second. According to Nielsen's program interruption limits (See section 7.2.1) this

Test case	Test case specification	Avg. program int. time	Max program int. time	Num. GC invocations
1	Chess, MC^2, 500k, 8k	0,53	3	6342
2	Chess, MC^2, 500k, 16k	0,66	3	2462
3	Chess, MS, 500k	44,71	52	67
4	Chess, MC^2, 1M, 8k	0,53	3	6342
5	Chess, MC^2, 1M, 16k	0,66	3	2462
6	Chess, MS, 1M	91,72	95	31
7	Regexp, MC^2, 500k, 8k	0,25	2	822
8	Regexp, MC^2, 500k, 16k	0,34	2	358
9	Regexp, MS, 500k	34,00	37	8
10	Regexp, MC^2, 1M, 8k	0,25	2	822
11	Regexp, MC^2, 1M, 16k	0,34	2	358
12	Regexp, MS, 1M	71,75	88	4
13	kXML, MC^2, 1M, 8k	1,44	40	41814
14	kXML, MC^2, 1M, 16k	1,84	44	21042
15	kXML, MS, 1M	1003,41	3097	183

Table 8.2: Program interruption results.

would not only spoil the user's sense of the program to react instantaneously, but also interrupt the person's flow of thoughts.

Also notice that the MC^2 results are equal for the Chess and Regexp benchmark with the same heap window size and different heap size. This happens because the MC^2 implementation will not allocate more windows than necessary on the heap to leave some room for large object windows or handle windows. This means that even though there is 1 MB available it will use below 500 kB of heap memory. The mark-sweep collector shows a different result when altering the heap size: both the average and maximum program interruption time seems to double when the heap size doubles. This happens because the mark-sweep implementation scans the whole heap in the sweep state.

The heap window size of the MC^2 collector does not seem to affect the program interruption time that much, but the number of garbage collection invocations are almost doubled with half window size.

This shows that the program execution time of the MC^2 implementation is much smaller than the program interruption of mark-sweep and therefore confirms the MC^2 article's statement:

> *"The pause times for MC^2 is 10-17 times lower than a copying garbage collector and 7-13 times lower than mark-sweep in a heap that is 1.8 times the program live size [SMB04]."*

It is now interesting to see if all the small program interruptions from the MC^2 implementation will yield a larger program execution time compared to running with the mark-sweep collector. This can be discovered by looking at the throughput of the benchmarks, covered in the next subsection.

8.4.2 Throughput

The throughput of the garbage collection implementations are tested by using the score from the EEMBC Java benchmarks. To produce a more comparable result the score from each benchmark will be normalized, so that the case with the highest score gets a normalized score of 1 and all the other a score between 1 and 0. The higher the throughput of the garbage collector, the smaller is the score and therefore also the program execution time.

The normalized throughput results from the chess benchmark (test case 1, 2, 3, 4, 5 and 6) are shown in figure 8.2. This figure shows that the score of the test cases are almost equal for the chess algorithm. The mark-sweep collector has the two highest scores and therefore the lowest throughput and the highest program execution time.

Test case	Test case specification	Normalized score
1	Chess, MC^2, 500k, 8k	0,98
2	Chess, MC^2, 500k, 16k	0,95
3	Chess, MS, 500k	1,0
4	Chess, MC^2, 1M, 8k	0,98
5	Chess, MC^2, 1M, 16k	0,95
6	Chess, MS, 1M	0,99
7	Regexp, MC^2, 500k, 8k	1,00
8	Regexp, MC^2, 500k, 16k	1,00
9	Regexp, MS, 500k	1,00
10	Regexp, MC^2, 1M, 8k	1,00
11	Regexp, MC^2, 1M, 16k	1,00
12	Regexp, MS, 1M	1,00
13	kXML, MC^2, 1M, 8k	0,48
14	kXML, MC^2, 1M, 16k	0,44
15	kXML, MS, 1M	1,0

A smaller score is better.

Table 8.3: Throughput results.

In the throughput results in table 8.3 one can see that the throughput does not differ much in the test cases, with the exception of mark-sweep in the kXML benchmark.

The reason why the kXML benchmark is so slow with mark-sweep is that all the small objects fragments the heap and makes the mark-sweep implementation traverse the free list very often and it has to insert all the collected objects into the free list. Because the kXML benchmark generates so many small objects this list can get long and it will therefore take some time to traverse the it.

All in all this throughput benchmark shows that the score in every test case is equal or lower for the MC^2 implementation. The lower score means that the MC^2 collector have a higher throughput and lower total program execution time than the mark-sweep implementation.

This means that the overhead from the MC^2 implementation is less than or equal to the overhead from the mark-sweep implementation in all test cases.

This confirms the MC^2 article's statement:

> *"We compared the performance of MC^2 with a non-incremental generational mark-sweep collector and a generational mark-compact collector, and showed that MC^2 provides throughput comparable to that of both of those collectors [SMB04]."*

8.5 Benchmark result summary

The MC^2 garbage collection implementation is now benchmarked against the mark-sweep implementation. The benchmarking confirmed the expected result from the theoretical comparison of the implementations: The MC^2 collector will produce a lot of small program interruptions and the mark-sweep collector will produce a few long program interruptions.

The throughput of the implementations was almost equal between the MC^2 and mark-sweep implementations, except in the kXML benchmark where mark-sweep had to insert every unused object into its free list.

All the tasks done according to the project assignment are now done. This report continues with a final chapter, describing the results, project experiences and possible future work.

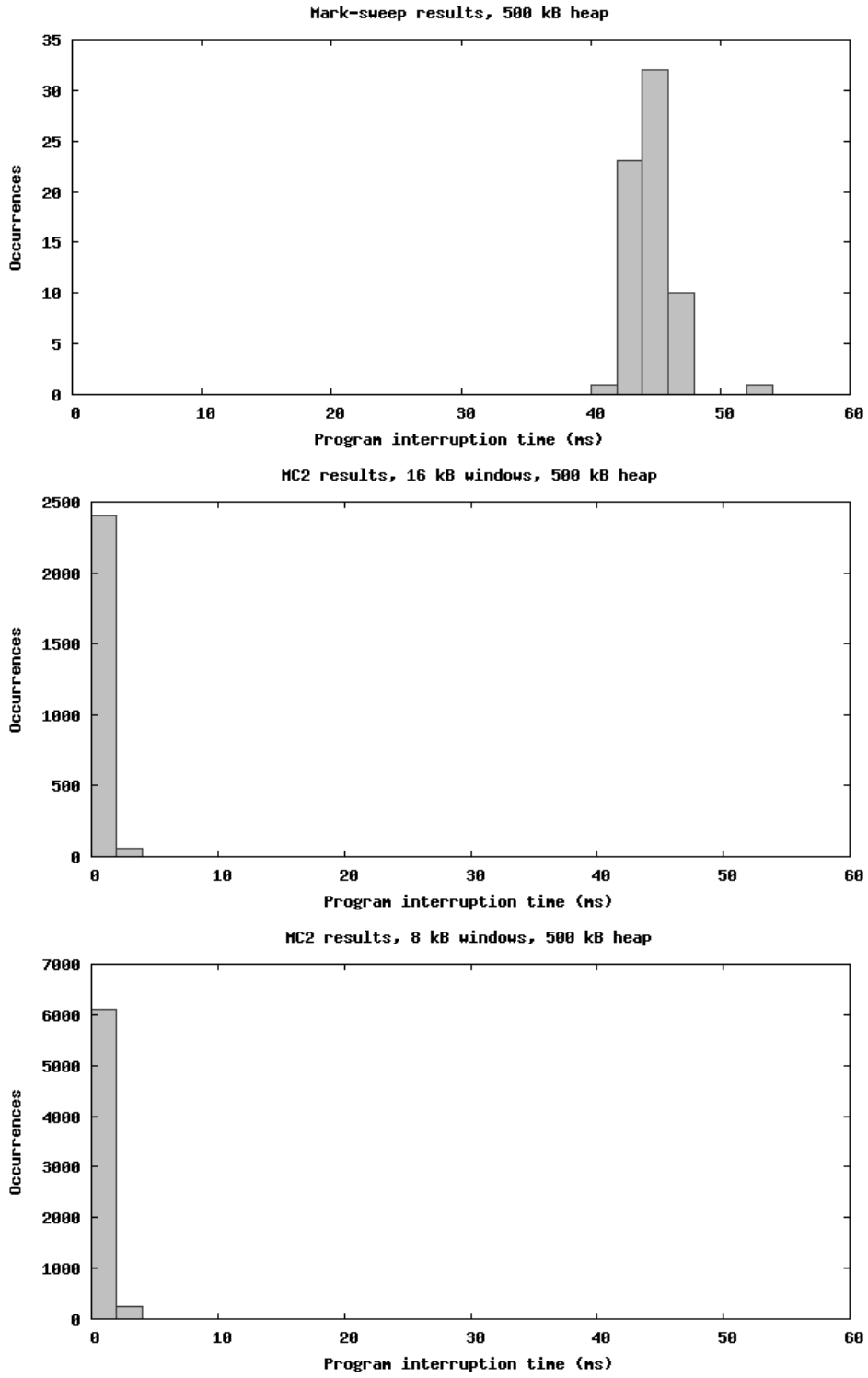
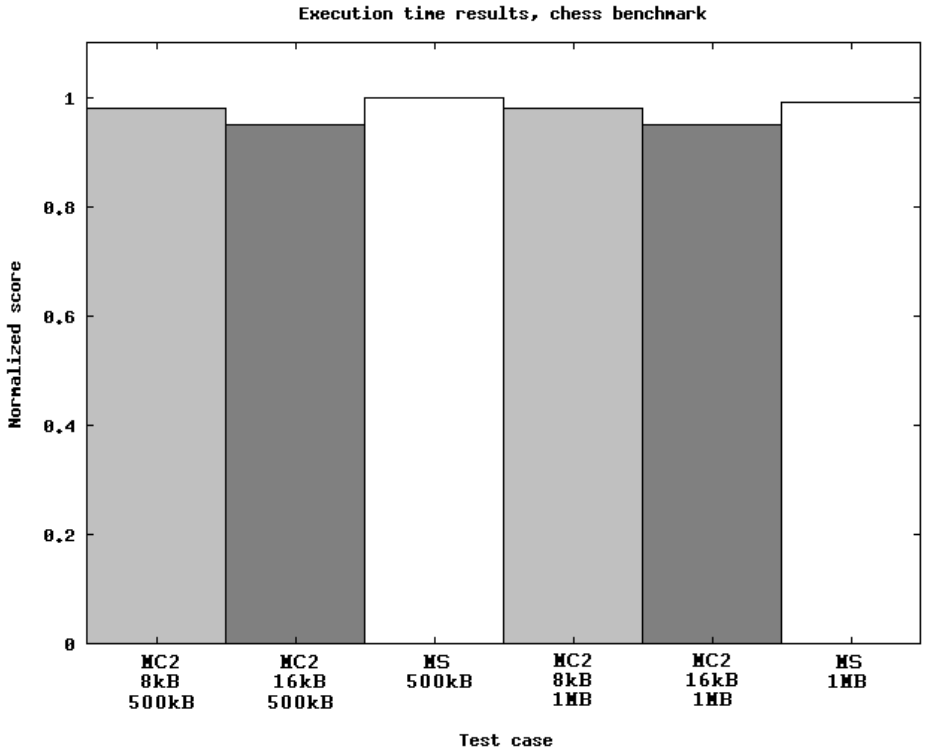


Figure 8.1: Chess benchmark test results with 500 kB heap.



A smaller score is better.

Figure 8.2: Throughput result from the chess benchmark.

Chapter 9

Conclusion

This chapter presents the results of the project and an evaluation of the project process itself. A list of possible enhancements for the garbage collectors are also included.

9.1 Main contributions

This project has led to many changes in the AVM and some changes in the microcontroller hardware, besides the implementation of the garbage collector. The main results are:

- The MC^2 algorithm has been implemented.
- The MC^2 article's results have been confirmed.
- The mark-sweep algorithm is working again.
- The AVM has been improved.
- Hardware bug found.

The results are presented in greater detail in the following subsections.

9.1.1 The MC^2 algorithm has been implemented

The AVM now includes an implementation of the MC^2 algorithm that has been tested functionally (see chapters 5 and 6). This implementation creates a shorter program interruption time for the user's Java programs than the earlier implemented mark-sweep implementation.

This implementation is the first MC^2 implementation that is run on a real microcontroller with dedicated hardware to support it.

9.1.2 The MC^2 article's results have been confirmed

The MC^2 algorithm has now been implemented and tested in a microcontroller running with limited processing power and main memory. The benchmarking of the implementation shows that it outperforms the mark-sweep implementation in program interruption time and has a throughput that is equal or better than mark-sweep (see chapter 8).

This confirms the MC^2 article's statement about program interruption time:

> *"The pause times for MC^2 is 10-17 times lower than a copying garbage collector and 7-13 times lower than mark-sweep in a heap that is 1.8 times the program live size [SMB04]."*

It also confirms the statements about throughput:

> *"We compared the performance of MC^2 with a non-incremental generational mark-sweep collector and a generational mark-compact collector, and showed that MC^2 provides throughput comparable to that of both of those collectors [SMB04]."*

9.1.3 The mark-sweep implementation is working again

The rearrangement of the object structure (see section 4.2.1) and the implementation of threads made it necessary to fix the mark-sweep implementation. The mark-sweep collector has been fixed (see section 4.2.3) and is now working again in the AVM and has been functionally tested and benchmarked with the same programs as the MC^2 implementation.

The user of the Atmel Virtual Machine now can choose whether to use the mark-sweep or the Memory-Constrained Copying implementation by editing the config file.

9.1.4 The AVM has been improved

The AVM has been modified to support an efficient MC^2 implementation (see section 4.2). These modifications have made the AVM closer to the JVM specification [LY99] because arrays are now a subclass of the type `java.lang.Object`.

The AVM is now safer to use because objects and handles are allocated by a cleaner interface that ensures correct initialization. This prevents many errors due to incorrect object initialization.

When benchmarking the implementations many errors were found due to the fact that the AVM never had run such complex programs before. The EEMBC Java benchmarks had been run before, but in a stripped down version because of the large memory requirement. One example of such an error was in the Java object lock system:

Java objects can be locked when accessing them to prevent that other threads access the object at the same time. During the first benchmark run the AVM ran out of C heap space, due to the locks never being deallocated after use (An object lock is allocated in the C heap of the main memory).

9.1.5 Hardware bug found

A hardware bug was found after the changes in the microcontroller architecture that broke the special write barrier functions that only the MC^2 implementation uses. This bug was identified and fixed. For a more detailed description of this bug see section 9.2.1.

If this bug had not been found before the microcontroller was realized in silicon the cost of repairing it would have been enormous compared to finding and fixing it in this project.

9.2 Project Experiences

This section contains a section about implementation challenges. A short evaluation of the process flow in this project is also included.

9.2.1 Implementation challenges

This section contains the main implementation challenges that was encountered during this project.

Few Mistral boards

During the development of the garbage collector there has been a shortage of Mistral boards. At most Atmel had three Mistral boards for use. One of these was always occupied for module verification and one of them was occupied for Java debug protocol implementation. The third Mistral board was a few times brought to customers for a week at a time and was therefore unavailable.

This led to a shortage of development boards and a halt in the development. Fortunately the Java Debug board could be borrowed for short periods and the implementation could continue.

The Atmel Virtual Machine

Developing a garbage collector for the AVM was a bit of a challenge. The AVM is still under development and is not totally compliant with the J2ME CLDC 1.1 specification. It has not been tested very thoroughly, so one can expect to find a few errors when running complex Java programs that have never been run on the AVM before.

For instance there was a bug in the method invocation code that made the current frame overwrite an object lock because of an addition error. This error only appeared in specific cases depending on the number of arguments that were passed to the methods invoked. The lock error did not appear at once, but after a long time of execution. This made it difficult to locate the bug, but once it was found it was easily fixed.

When an error appears in the Java execution it is not given that the error stems from the source code. It can also stem from the C compiler, the assembler,

the microcontroller hardware or the Vitra/Pathfinder system. The most likely source of errors is in the software running on the controller and this source should be thoroughly checked before looking for other sources.

Changing the nursery collector

During the implementation of the MC^2 algorithm it was first decided that the copying algorithm should be used as the nursery collector garbage collection algorithm. This algorithm was implemented and tested and was working properly.

When the old generation marker was implemented it was clear that the copying algorithm would not work as a nursery collector because it could not guarantee that the live objects from the nursery would fit in the old generation before copying. The mark stack of objects in the old generation would also have been destroyed when the old generation collector was invoked. Another nursery collector therefore had to be implemented.

The mark-copy collector was the next logical choice and was implemented very rapidly. The reason for the rapid implementation was that the copying implementation had sorted out some bugs in the allocator code and that the mark-copy used the Deutch-Schorr-Waite marker that had been implemented in another version for the mark-sweep collector.

All in all the change of nursery collector delayed the project by two days approximately.

Implementing and debugging a memory management system

When implementing a memory management system, like the MC^2 is part of, it is crucial that every memory reference is correct when returning control to the user program.

If the object graph in a Java Virtual Machine is not equal before and after garbage collecting errors *may* appear in the user program. Sometimes the errors are insignificant and do not crash the execution, like when every "2"-digit in the program output disappears and the program continues (This case was encountered while debugging the MC^2 implementation.), but most of the time the errors crash the program after a while.

When the effects of an error is observable a long time after the error has happened, it is difficult to find the error. Pathfinder's data- and program trace- and data watchpoints (See section 6.1.3) was very handy when debugging these errors.

Changes in the microcontroller architecture

In January 2005 the pipeline of the microcontroller was redesigned to make it more efficient. The instruction set architecture was also updated and the assembler for the microcontroller was updated. These changes was completed in week 14 and a new bit-file was ready for the Mistral prototype board.

Because of the instruction set changes the trap library and some other assembly files had to be modified and tested. In addition to this a hardware bug was found in the `aputfield_quick` and `aastore` instructions. When these instructions were executed they stored the next instruction address (normally stored in the link register) in the Java Barrier Configuration Register (JBCR) when the write barrier was violated. This made the microcontroller execute the same instruction one more time and destroyed the Java operand stack and the object or array the pointer should be stored in.

A test case was produced that confirmed and located the hardware bug and a fix could be made in the RTL-code. The microcontroller design with the hardware fix was then scheduled for synthesis the next night. The first synthesis stopped because the Mistral FPGA was too small for the design. Some I/O modules was removed and the synthesis was rescheduled to the next night. The next morning the second synthesis was found unusable during testing because of a misplacement of the instruction cache. The design was fixed and the third synthesis worked perfectly and the situation was back to normal again.

The new instruction architecture set and the hardware bug caused the testing of the MC^2 implementation to stop for about a week.

9.2.2 Status meetings

During this project weekly status meetings were held at Atmel with Morten Haaker and Lars Even Almås. During these meetings the project status was compared to the project plan and the recent changes in report were reviewed.

Lars Even also took part in the many technical discussions outside the status meetings. His opinions and suggestions were a valuable contribution to the design and implementation of the garbage collection algorithm and support functions.

9.2.3 The project schedule

The project schedule is shown in figure 9.1, and contains the following tasks:

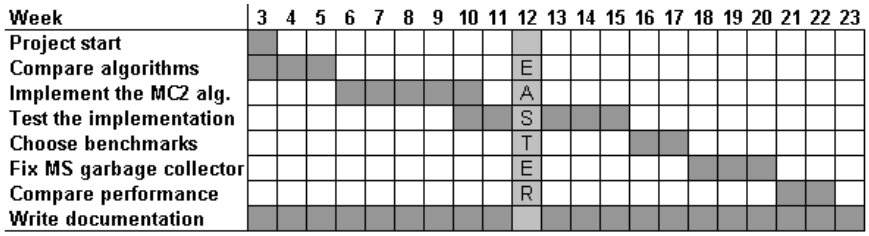


Figure 9.1: Project schedule.

Project start Make the project schedule, create a report document and a disposition.

Compare algorithms Compare the mark-sweep and the MC^2 algorithm against each other.

Implement the MC^2 algorithm Implement the MC^2 algorithm in the Atmel Virtual Machine.

Test the implementation Test the MC^2 implementation functionally.

Choose benchmarks Choose a set of test programs to analyze the performance of the implementations.

Fix MS garbage collector Changes made in the AVM after the mark-sweep algorithm was implemented has made it necessary to fix the mark-sweep implementation to make it work again.

Compare performance Benchmarking the implementations and compare the performance.

Write documentation Write the project report (this document).

During this project the project schedule was followed closely. Every task was finished when expected, except for the fixing of the mark-sweep algorithm. This task was completed one week earlier and therefore more time was spent benchmarking.

One of the reasons why the mark-sweep algorithm was done so quickly was that the Deutsch-Schorr-Waite marker was already implemented and tested as nursery marker for the MC^2 algorithm.

The gain in time was very handy, since the EEMBC Java Benchmark suite proved to be hard to get running in the AVM. The complexity of these benchmarks also led to only three of six benchmarks was running on the AVM before the project end.

9.3 Future work

This section describes the work that can be done with the garbage collector in the future.

9.3.1 Optimize the MC^2 implementation

Even though the MC^2 implementation clearly outperforms the mark-sweep implementation in program interruption time there is still some room for optimization.

One of these optimizations is to improve the garbage collectors handling of static variables. The static variables are scanned entirely in every nursery garbage collection and marking increment. Scanning of the static variables means that every static field in every class structure must be visited and checked. This could be avoided by making the `putstatic` bytecode tell the garbage collector when a static variable is overwritten. The garbage collector could then check if the object was in the nursery, and give it a nursery mark, or if it was in the old generation and the garbage collector was in the marking state.

This optimization requires that the `putstatic` instruction is trapped and run in software and not executed directly in hardware. This would make the execution of the Java programs slower. The `putstatic` command is not the most frequently used bytecode so it must be tested if the gain in the garbage collection execution time outweighs the loss in Java execution time.

9.3.2 Make the garbage collectors throw an OutOfMemoryException

At the project end both the MC^2 and the mark-sweep garbage collector terminates the AVM with an error code when the heap is full. According to the Java Virtual Machine specification a OutOfMemoryException should be thrown when this happens.

This makes the Java program capable of catching the exception and continue execution or exit properly. To make this possible a method should be created that creates a new out of memory exception and pushes this object on the Java stack, returns to the Java program and throws the exception.

9.3.3 Test the MC^2 implementation against the remaining benchmarks

Because only the chess, regular expression and kXML benchmark of the six EEMBC Java benchmarks were running at the project end the remaining (crypto, parallel and png) must also be tested with the garbage collection implementations.

This would also test the garbage collection implementations with several threads running. This test has been excluded because the thread package had not been fully implemented at the project end.

The kXML benchmark also needs to be tested with a small heap. At 500 kB the MC^2 implementation encounters many problems mainly because of fragmentation of the heap window structure. The heap must therefore be defragmented on a window level to ensure that the Java program can continue to run.

9.4 Conclusion summary

A theoretical comparison of the Memory-Constrained Copying algorithm and the mark-sweep algorithm was done. This comparison showed that in theory the MC^2 algorithm should outperform the mark-sweep algorithm with respect to program interruption time.

The lower program interruption encouraged the implementation of the algorithm and the AVM was modified to support the algorithm before it was implemented and functionally tested.

To test the MC^2 implementation against the mark-sweep implementation a set of benchmarks was chosen. The best candidate was the EEMBC Java benchmark suite.

The benchmarking showed that the MC^2 implementation produced a much lower program interruption than the mark-sweep implementation. The throughput of the algorithms were almost equal.

Garbage collection is a nice aid for programmers as it makes certain program errors, like dangling pointers, impossible. The cost of the garbage collection can be very large, at least in an embedded setting where there is a limited amount of processing power.

It is often being discussed whether Java is suitable for embedded systems [Nis05]. The big issue here is performance, both because Java is an interpreted language and because it has to collect garbage. Both these factors create an overhead that can be too much for a microcontroller with already limited powers.

The new microcontroller from Atmel has hardware support for interpreting the Java bytecodes and is using the new Memory-Constrained Copying algorithm for garbage collection. These enhancements makes the execution of Java programs very fast and creates a low program interruption. This makes the new Atmel microcontroller able to run interactive programs without large interruptions.

Bibliography

[Amu04] Kai Krisitan Amundsen. *Garbage collection and memory management in a Java Virtual Machine for an embedded system.* NTNU, 2004.

[Ash05] *Pathfinder datasheet.* World Wide Web, http://www.ashling.com/datasheets/DS201-32-PathFinder.html, Visited 15/04 2005.

[Atm03] *AVR ATmega8 datasheet.* Atmel Norway, 2003.

[Atm04a] *Internal document: AVR-II CPU arcitecture, revision 0.96.* Atmel Norway, 2004.

[Atm04b] *Internal document: JVM Documentation, version 1.7.* Atmel Norway, 2004.

[Bri05] *Algorithms and pseudo-code.* World Wide Web, http://userpages.wittenberg.edu/bshelburne/Comp150/Algorithms.htm, Visited 02/06 2005.

[Dua98] Lawrence A. Duarte. *The microcontroller beginner's handbook.* Prompt Publications, 1998.

[EEM03] EEMBC. *EEMBC Benchmarks for the Java 2 Micro Edition J2ME Platform, White Paper.* EEMBC, May 2003.

[EEM05] *EEMBC Java Benchmark suite home page.* World Wide Web, http://www.eembc.org/Benchmark/java.asp, Visited 21/04 2005.

[EK05] John Ellis and Pete Kovac. *GCBench.java.* World Wide Web, http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/applet /GCBench.java, Visited 15/04 2005.

[HB97] Ken Hines and Gaetano Borriello. Dynamic communication models in embedded system co-simulation. In *Design Automation Conference*, pages 395–400, 1997.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification, second edition.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[McC93] Steve McConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, Redmond, WA, USA, 1993.

[mem05] memorymanagement.org. *The memory management glossary: garbage collection*. World Wide Web, http://www.memorymanagement.org/glossary/g.html#garbage.collection, Visited 15/04 2005.

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., 1993.

[Nis05] Ed Nisley. *Java: Stirring the Cup*. World Wide Web, http://www.ddjembedded.com/resources/articles/2002/0202l/0202l.htm, Visited 15/04 2005.

[Pen05] *CaffeineMark web site*. World Wide Web, http://www.benchmarkhq.ru/cm30/info.html, Visited 30/05 2005.

[PH90] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[Set89] Ravi Sethi. *Programming languages: concepts and constructs*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[SMB04] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. MC^2: High-performance garbage collection for memory-constrained environments. In *OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[SPE05] *SPEC JVM98 Benchmarks*. World Wide Web, http://www.spec.org/jvm98/, Visited 15/04 2005.

[Sta98] William Stallings. *Operating systems (3rd ed.): internals and design principles*. Prentice-Hall, Inc., 1998.

[SW67] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[Tho72] Lars-Erik Thorelli. Marking algorithms. *BIT*, 12(4):555–568, 1972.

[Top02] Kim Topley. *J2ME in a nutshell, a desktop quick reference*. O'Reilly, March 2002.

[VRG98] Narayanan Vijaykrishnan, N. Ranganathan, and Ravi Gadekarla. Object-oriented architectural support for a java processor. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 330–354. Springer-Verlag, 1998.

[Wei84] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.

[WR00] Russel Winder and Graham Roberts. *Developing Java Software*. John Wiley & Sons, Inc., 2000.