INSTITUTE OF COMPUTER SCIENCE
AALBORG UNIVERSITY

Fredrik Bajersvej 7E 9220 Aalborg Ø, Tlf: +45 96 35 80 80 Fax: +45 98 15 98 89

Title:
Garbage Collection in a gbeta Virtual Machine with the Train Algorithm

Project Unit:
F10SE,
Feb. 1th 2000 – June. 18th 2001

Group:
E3211B

Group Members:
Peer Møller Ilsøe
Simon Hem Pedersen

Supervisor:
Erik Ernst

Copies: 4

Pages: 145

Abstract

In this project a virtual machine, gbvm, with a unique train algorithm garbage collector is implemented for the programming language gbeta.
The main focus of this thesis is memory management using the train algorithm and experiments with the implementation.
Firstly, this thesis introduces relevant theories and work including an introduction to gbeta, a overview of virtual machine architectures, and descriptions of garbage collection concepts, properties, and algorithms.
Secondly, the design and implementation details of gbvm are presented
Thirdly, experiments investigating the performance of both parameter but also algorithm changes of the train algorithm are conducted and discussed. The experimental framework implemented for this thesis allows for further experiments with gbvm.
Finally, it is concluded that it is difficult to find a fixed general setting, which is both time and space efficient with all the tested programs.

Resume

Dette afgangsprojekt dokumenterer udviklingen af gbvm – en gbeta virtuel maskine med en unik train-algoritme garbage collector. gbeta er en generalisering af programmingssproget BETA. gbeta tilbyder de samme faciliteter som BETA, men derudover er mere udtryksfulde abstraktionsmekanismer og større run-time fleksibilitet tilføjet.

I den nuværende implementation af gbeta er den virtuelle maskine inkorporeret i kompileren, hvilket giver et ineffektivt system, hvor det er *umuligt* at kompilere en gang og derefter afvikle mange gange uden genkompilering. Dette motiverer en seperat virtuel maskine, som gør det muligt at afvikle et program uden at gentage kompileringsfasen. I dette afgangsprojekt dokumenteres designet og implementationen af sådan en virtuel maskine kaldet gbvm.

Hovedfokus i dette afgangsprojekt er memory management ved hjælp af train algoritmen. En memory manager med en unik train algoritme garbage collector er designet og implementeret. Der er udført eksperimenter med gbvm, og det framework der ligger bag ved disse eksperimenter gør det muligt at fortsætte med flere eksperimenter. Eksperimenterne fokuserer på hvordan både ændringer af forskellige implementations-parametre, men også selve algoritmen, kan påvirke plads og tidseffektiviteten. Derudover sammenlignes gbvm med andre gbeta afviklingssystemer.

Indholdet af dette afgangsprojekt er opdelt i tre hoveddele. Den første del dokumenterer vores studier i materiale, som er relevant for dette afgangsprojekt. Her introduceres gbeta med dets forskellige modelleringsentiteter og byte kode format. Derefter præsenteres tre forskellige arkitekturer for virtuelle maskiner og til sidst forskellige aspekter af memory management.

Anden del dokumenterer design og implementation af vores virtuelle maskine modul og vores memory management modul. I disse kapitler er der undervejs knyttet evaluerende kommentarer, hvor vi har fundet det relevant.

I den tredje del dokumenteres en række eksperimenter med gbvm, og vi kommer frem til hvilken indvirkning introductory space og car størrelser i heapen og forskellige oprettelsesstrategier for nye trains har på tids- og pladsperformance.

Afsluttende beskrives relaterede virtuelle maskiner og kommende arbejde. Vi konkluderer, at det at svært at finde en god kombination af introductory space og car størrelse, og at den nyeste af de to foreslåede oprettelsesstrategier for nye trains er den bedste.

Preface

This report is a master's thesis in computer science, programming systems. The report is directed towards people with interests in object oriented programming languages, virtual machine design and implementation, and especially memory management using the train algorithm.

A complete bibliography is located in the back of the report. References to the bibliography are made with square brackets e.g., [Ern99, p42] which refers to the Ph.D. thesis "gbeta - a Language with Virtual Attributes, Block Structures, and Propagating, Dynamic Inheritance" page 42. The references are not allways annotated with page numbers.

References to figures are made as (see figure x.y), where x represents the chapter and y is a consecutive number in that chapter. The same applies to tables.

Some typography in the text is used to clarify the meaning. When a new concept is introduced the word is typeset *new concept*, class names are typeset `ClassName`, and methods, attributes, and other things referring to code are typeset `method`. When a variable is a pointer to a class (`ClassName`), its type is abbreviated `classNamePtr`. An index in the back shows where in the report new concepts are introduced and explained.

We have included a CD-ROM with this thesis that contains: the source code, the graphs of the experiments, and the data which form the basis for these graphs. A file called `README` on the CD-ROM further describes the contents of this CD-ROM.

We would like to thank Ricki Jensen, Christian Jørgensen, and Michael Wojciechowski for letting us use their benchmark programs and gbeta virtual machine for comparison.

Peer Møller Ilsøe   Simon Hem Pedersen

Contents

1. Introduction

This master's thesis is devoted to three main areas: The programming language gbeta, implementation and design of virtual machines, and memory management. Memory management and especially the train algorithm is the main focus.

gbeta was developed by Erik Ernst, Assistant Professor at Aalborg University. It is a modern object oriented programming language, which implements a superset of the programming language BETA. gbeta combines static type safety with dynamic inheritance, i.e., it combines the design goals safety and flexibility. gbeta has a higher level of run-time flexibility compared to BETA. For instance it is possible at run-time to combine two methods into a new method, and one can dynamically change the class of an object. The increased run-time flexibility combined with static type safety in gbeta increases the complexity and requirements for the compiler and the run-time system.

So far the work put into the gbeta project has concentrated on the design and implementation of the front-end of the language, i.e., the gbeta compiler. With respect to the back-end, i.e., the run-time system including, e.g., a virtual machine, automatic memory management, etc., a large amount of research and development still has to be done, but [JJW01] and this thesis are a start.

Automatic memory management is a optional part of a run-time system. If no automatic memory management system is present, the application programmer has to incorporate complicated and error-prone memory management into each application developed – if we assume that he does not want to just use memory until it depleted. Automatic memory managers solves the problem by ensuring that objects no longer needed are reclaimed.

Many different memory management strategies have been proposed. One of them is the train algorithm. The main purpose of the train algorithm is to ensure low mean time disruptions while still eventually reclaiming all unneeded objects.

In [IP01] we implemented a memory management component that used our first implementation of the train algorithm. We also implemented the core of a virtual machine which used the memory management component, but they were not working well when joined.

The evaluation of the first virtual machine implementation gave us reason to change the design and end up with a more functional virtual machine, which actually had safe points between each instruction execution.

The evaluation of the first implemented memory management component also gave reason to a redesign and reimplementation of a new and improved train algorithm garbage collecting memory management system.

1.1. Hypothesis

Other systems with a train algorithm garbage collector use a one- or even n-generation copy collector in the generation before the generation with train algorithm (see chapter 8). We would like to explore what happens, if no such copy collector is present. A simpler approach could both be easier to maintain and debug but most importantly, it could also yield better performance. Our idea is that the train algorithm extended with an extra special car (called introductory space) is enough to yield both good performance and relatively low disruption times.

1.2. Contributions

The main contribution is our investigation of the train algorithm and how it might be set up differently. This includes the unique experiments conducted on our train algorithm memory manager, and the framework used to conduct these experiments. Since we have used scripts to conduct the experiments and create the resulting graphs, it is possible to further experiment using the framework developed as a part of this thesis.

Another contribution is the virtual machine component. Although the virtual machine component has not been the main focus in this thesis, we have implemented a competitive virtual machine capable of executing most gbeta programs without repetitions and concurrency.

1.3. Thesis Structure

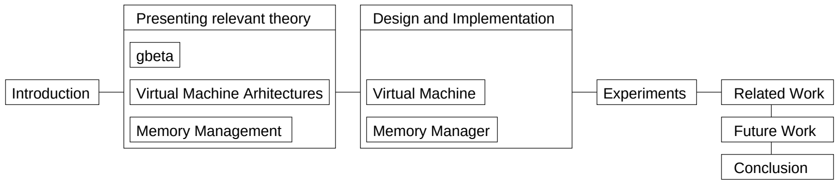


Figure 1.1.: *The structure of this thesis*

Our thesis structure is illustrated in figure 1.1. After this introduction we present relevant theory about the areas: gbeta, virtual machine architectures, and memory management. We present both the gbeta language and its byte code format. Then three different virtual machine architectures are presented and briefly compared. The theory part ends with a presentation of memory management concepts, properties, and a number of different garbage collection algorithms including the train algorithm is presented with their advantages and disadvantages.

We then present the design and implementation details of the virtual machine component (see chapter 5). In this chapter we describe the architecture of the virtual machine, describe interfaces, and explain how we have redesigned the virtual machine to allow for copy collection during both normal instruction execution and attribute initialization. In the end of the chapter we describe how the instructions were implemented, especially the multi-line instructions and complex single-line instructions. Finally we conclude with a short evaluation of the current implementation.

The design and implementation details of the memory management component are then presented in chapter 6. We present the unique heap layout, the architecture of the component, the data structures, the most important classes, and conclude with a description of our version of the train algorithm and its write barrier. Then we describe the interface of the component. At last we describe how we used different debugging techniques and tools to make the tedious task of debugging the whole system (called gbvm) more endurable.

Experiments with gbvm, and in particular the memory management system is presented in chapter 7. In this chapter we investigate the interrelation between different introductory space and car sizes in our heap, and time and space performance. Then we experiment with two different new train creation policies and compare their effects on performance. We have also conducted experiments that quantifies the time used in our write barrier, and other experiments that compare gbvm with other gbeta executing systems and with the Java HotSpot virtual machine. To give an idea of how much time is spent in the different parts of gbvm, we have also profiled it using `gprof` [GKM82].

The thesis ends with a chapter describing related work. In this chapter we focus on related run-time systems that use the train algorithm. Before the conclusion (see chapter 10) we present proposals for future work (see chapter 9).

You can read this thesis in several ways. The best way is of course to read it all, but if you are familiar with the theory presented, i.e., the gbeta language, virtual machine architectures, and memory management, you may skip the chapters 2, 3, and 4. As the main focus in this thesis is memory management using the train algorithm, it is also possible to skip chapter 5, if you are not interested in details of our virtual machine component. We do not recommend skipping chapter 5 though, since an understanding of the virtual machine component is important to fully understand the memory management component and the experiments conducted in chapter 7.

2. gbeta

gbeta is a generalization of the programming language BETA. gbeta generalizes BETA in two main areas namely more expressive abstraction mechanisms, and an improvement of the run-time flexibility without compromising the type safety. That gbeta is a generalization of BETA means that it implements a superset of BETA so every BETA program is also a gbeta program but in gbeta it is possible to write programs that exploit the extra features in gbeta not supported in BETA.

One of the extra features of gbeta is the possibility to do object metamorphism i.e., it is possible at run-time to take an existing object and modify its structure until it is an instance of a given class. Another extra feature in gbeta is the possibility to define relations between classes and in this way define a constraint graph of classes. The constraint graph ensures that these relations hold implying that one inheritance operation may give rise to a propagation of type changes in a framework of classes.

Since gbeta is an ongoing research project some parts of this chapter may be obsolete. In this chapter we introduce the language and the entities that a run-time system, supporting execution of gbeta programs, must handle. Then a second section describes the gbeta byte code format and the special way of addressing using run-time paths.

2.1. gbeta Entities

Figure 2.1 is a class diagram of the entities of a gbeta environment. In this section they will be described shortly in turn. For a more thorough explanation of gbeta entities and syntax see [Ern00] or [Ern99].

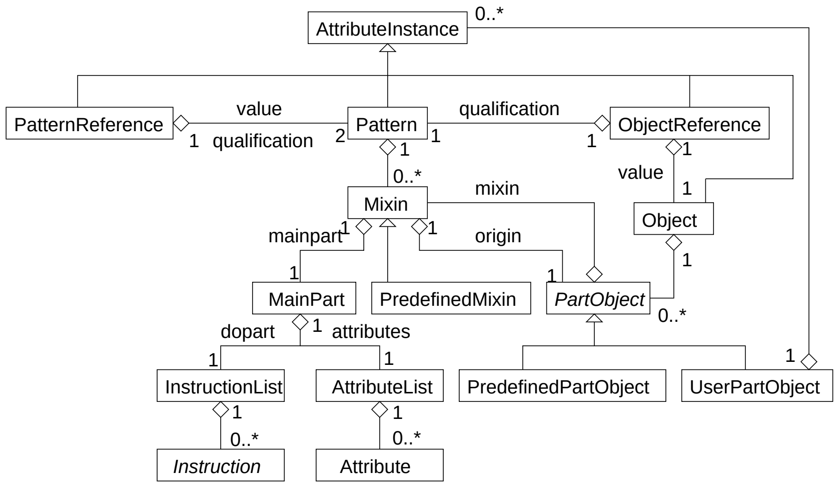


Figure 2.1.: *A class diagram of the gbeta entities to be handled by the virtual machine.* `AttributeInstance` *is super class for the classes* `Object`*,* `ObjectReference`*,* `Pattern`*, and* `PatternReference`

2.1.1. Mixin

A gbeta pattern is divided into mixins. A *mixin*, in a pattern, describes how to create one of the part object(s) in an object instance of the pattern.

A pattern consists of several mixins if specialization is used. The pattern `r` described in the syntax below consists of three mixins because it is a specialisation of `q` which again is a specialisation of `p`. So, one mixin is the difference between a pattern and its superpattern as long as only single inheritance is used. Returning to the example, the substance described in the main-part after the `p` in the definition of `q` is the mixin describing the difference between the pattern `q` and its superpattern `p`.

```
p : (# #);
q: p(# a: @integer #);
r: q(# #);
```

2.1.2. Pattern

A gbeta *pattern* handles every aspect of structure description. Every time substance (i.e., an object) is created in gbeta it is created according to some gbeta pattern. A gbeta pattern also specifies a run-time context namely its enclosing part object(s). Two patterns with the same syntax can therefore be different because they have different origins i.e., enclosing part object(s).

Because different patterns can have a different number of mixins, their size vary.

Syntactically a pattern is described as:

<Name> : <Merge>

Where <Name> is an identifier and <Merge> is a main-part, an identifier, or a `s & t & ... & z`-like expression. Here `s & t` is the pattern resulting from a *pattern merge* of the to patterns `s` and `t`. In figure 2.2 the patterns `s` and `t` are merged. The figure also illustrates the concept subpattern. A pattern `u` is a *subpattern* of a pattern `s` if and only if the mixin list of `s` can be obtained by removing zero or more mixins from the mixin list of `u`. In the figure the pattern `s & t` is a subpattern of both the pattern `s` and the pattern `t`. Correspondingly, a pattern `u` is a *superpattern* of a pattern `s` if and only if the mixin list of `s` can be obtained by adding zero or more mixins to the mixin list of `u`.

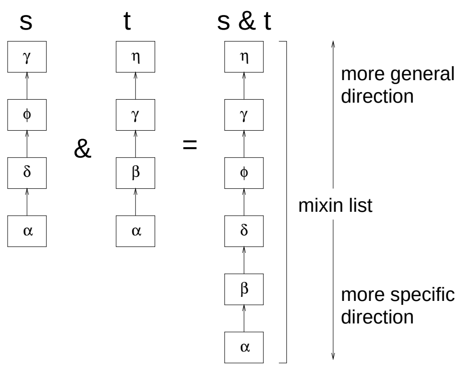


Figure 2.2.: *Illustration of pattern merging (`s & t`) between the patterns `s` and `t`*

The merge algorithm can be described as the function `merge(s,t)` which takes two patterns and returns a pattern merge of them if possible. In the pseudo code of `merge(s,t)` in figure 2.3, the `lowest(s)` function returns the most specific mixin of the pattern `s`, which has not yet been used in the merge operation. `u` is the result of the pattern merge of `s` and `t`. Moving adds in the general end of the resulting pattern.

```
merge(s,t){
  while(s not empty or t not empty){
    if(lowest(s) == lowest(t){
      move lowest(s) to u
      remove lowest(t)
      }
    else if(lowest(t) ∉ s move lowest(t) to u
    else if(lowest(s) ∉ t move lowest(s) to u
    else give up
  }
  return u
}
```

Figure 2.3.: *The merge algorithm*

Below is an example of a specification of a pattern `p` with a nested pattern (`nestedEmptyPattern`):

```
p: (# nestedEmptyPattern: (# #)
   #);
```

2.1.3. Object

Objects are instances of patterns. They are described syntactically with an `@`:

<Names> : @ <Merge>

Where <Names> is a comma-separated list of one or more identifiers. If e.g., an instance of the pattern `r` (see subsection 2.1.1) is wanted, it is written in gbeta as:

```
instanceOfPatternR : @r;
```

An object consists of a number of part objects, one for each mixin in its pattern, so an instantiation of the pattern `r` results in an object (`instanceOfPatternR`) with three part objects.

2.1.4. Part Object

Part objects are instances of mixins. A part object consists of a number of attributes, one for each attribute in the main-part of its mixin. Because part objects can contain different numbers of attributes, different part objects can have different sizes.

2.1.5. Main-Part

A *main-part* is the main piece of syntax used to construct gbeta programs. It consists of the delimiters `(#`, `#)` and four optional parts:

```
(#  <attributes>
enter  <evaluation>
do  <imperatives>
exit  <evaluation>
#)
```

where <attributes> is a list of attributes each syntactically described as:

```
<Name>   :  <Clarifiers>    <AttributeDenotation>
```

The <Clarifiers> can be one or a combination of the reserved characters: ' ', @, |,^and ##. The <Attribute-Denotation> specifies the qualification of the attribute, i.e., it is a type constraint on the attribute when one of the clarifiers^,^| , or ## is used. It is more like an initialisation expression for non-variable attributes expressed with the ' ', @, and @| clarifiers.

The *enter-part* can be compared to the initialization part of an object's constructor in other object oriented languages or the parameters of a method. It describes how many parameters this main-part instantiated into a method invocation takes when executed, and it also defines the semantics of value assignment to an object instance of this main-part.

The *do-part* can be compared to the body of a method, i.e., the code an instantiation of this main-part executes.

Finally, the *exit-part* can be compared to the specification of what is returned by a method. It describes what is returned when a method invocation instance this main-part is executed. The exit-part also defines the semantics of value extraction from an object instantiation associated with this main-part.

The gbeta compiler compiles main-parts into a list of attributes containing initialisation byte code instructions for each attribute and a list of byte code instructions corresponding to the do-part. The enter- and exit-part are present as byte code instructions in places in the byte code of the program where the main-part is used.

2.1.6. Pattern and Object References

To make an attribute in a main-part a reference to an object the^character is used and to make it a reference to a pattern, ## is used. For instance, `object_ref` is an object reference to an integer object and `pattern_ref` is a pattern reference to the pattern `p` in the following example:

```
object_ref  : ^integer;
pattern_ref : ##p;
```

2.2. gbeta Byte Code

gbeta byte code differs from other types of byte code like p-code [Nel79], java bytecode [LY97], and Smalltalk bytecode [GR89], in more than one way. First of all it is not really *byte* code in the sense that each opcode is a byte. It is more like a high-level human readable ascii assembly language. Secondly, it has got multi-line instructions with block structures that can be nested instead of labels and jumps. Thirdly, addressing is based on run-time paths.

2.2.1. Byte Code File Format

A gbeta byte code file is identified by its .gbc filename suffix and the contents are built from the following grammar. We have omitted the actual instructions since there are too many to mention, and the point here is the structure of the file rather than the instructions. The instruction will be discussed in the later subsections 2.2.3, 2.2.4, and in chapter 5 several of them are described more thoroughly. Line breaks are important for parsing a gbeta byte code file, but are omitted in the grammar for clarity. <filepath> is a standard Unix file path and it is only included when the source program has been divided into more than one file.

```
  <byteCodeFile>  ::=   <mainPart> *
      <mainPart>  ::=  'MainPart("' <mainPartId> '"' <attribute> *'|' <doPart> ')'
    <mainPartId>  ::=  '‘' ( <filepath> ':')? <int>
     <attribute>  ::=  '"' <nameAndIndex> '": (' <instruction> *')'
   <instruction>  ::=   <singleLineInstruction> | <MultiLineInstruction>
        <doPart>  ::=   <instruction> *
 <nameAndIndex>  ::=   <name>  '/'  <int>
          <name>  ::=  [a-z A-Z_] [a-zA-Z0-9_]*
           <int>  ::=  [0-9]+
```

Bytecode File Example

An example of a gbeta byte code file (remoteInsertBug.gbc from the nodist directory of the 0.81 distribution) is in figure 2.4. We will not explain what the program actually does – the important thing here is the structure of the code.

```
MainPart("‘176"
   "x/0": (
      PUSH-ptn_"object"
      ADD-mainpart ‘70 origin
      NEW,_ptn->obj
      INSTALL-obj
   )
|
   PUSH-ptn "x/0","p/0"
   ADD-mainpart ‘148 origin
   NEW,_ptn->tmp 1
   CALL tmp(1)
   RESETFRAME
)
MainPart("‘70"
   "p/0": (
      PUSH-ptn_"object"
      ADD-mainpart ‘44 origin
      INSTALL-ptn
   )
|
)
MainPart("‘44"
|
   INNER 0
)
MainPart("‘148"
|
   PUSHI-string "Hello, world!"
   stdio/out
   RESETFRAME
)
```

Figure 2.4.: *Example of gbc byte code file (remoteInsertBug.gbc from 0.81 gbeta distribution)*

A byte code file is executed by executing the first gbc-main-part in the file which is the result of compiling the outer most gbeta main-part. This involves creating a mixin, a pattern, and instantiating the pattern before the do-part is executed.

2.2.2. Run-Time Paths

The gbeta byte code uses *run-time paths* to access patterns, objects, and part objects in the gbeta run-time system. Evaluation of a run-time path is always performed in context of a current-part-object. The abstract syntax definition for the run-time path, <rtp> , is given in the following:

```
           <rtp>  ::= '{' <step> (',' <step> )* '}'
          <step>  ::= <out> | <up> | <down> | <lookup> | <lookupIndirect> | <temp>
           <out>  ::= '<-' <int>
            <up>  ::= '^' <mainPartId>
          <down>  ::= 'v' <mainPartId>
        <lookup>  ::= '"' <nameAndIndex> '"'
<lookupIndirect>  ::= ''' <nameAndIndex> '''
          <temp>  ::= 'temp(' <int> ')'
    <mainPartId>  ::= '`' ( <filepath> ':')? <int>
  <nameAndIndex>  ::= <name> '/' <int>
          <name>  ::= [a-z A-Z_] [a-zA-Z0-9_]*
           <int>  ::= [0-9]+
```

In the following two subsections we will clarify the semantics of these run-time steps. There are two fundamental kinds of run-time path traversals. The first kind, which is used in the instructions `INNER` and `ADD-mainpart`, consists of steps which start with a step that returns a part object. The following steps operate on a part object from the previous step and return a part object either for the next step or the final result. We call these steps *part-object-steps* and these will be discussed in the following subsubsection.

The other type, which is used by the other instructions utilizing run-time paths, varies slightly in that it terminates its run-time path traversal with a special run-time step evaluation. We call such steps *last-steps* and discuss them further in the subsubsection following the part-object-steps. The steps prior to the last-step are evaluated like the part-object-steps.

Part-object-steps

`out-step (<-` <int> `)` goes to the surrounding part object <int> number of times starting from current-part-object.

`up-step (^`<mainPartId>`)` searches for the part object with <mainPartId> in the "more general" direction of the part object list, starting with the current-part-object.

`down-step (v` <mainPartId>`)` is like the up-step but search for the part object is in the "more specific" direction instead.

`lookup-step ("` <nameAndIndex>`")` takes the current-part-object and selects the attribute placed at the given index. This attribute is statically known to be an object. The most specific part object of the that object is selected. (<name> is debug info)

`lookupIndirect-step ('` <nameAndIndex>`')` chooses the attribute at <int> which is statically known to be an object reference. The most specific part object of the referenced object is selected. If the value of the object reference is NONE, a run-time error is raised and the currently executing gbeta-thread is killed.

`temp-step (temp(` <int>`))` takes the current frame on the temp stack and returns the most specific part object in the object at <int> relative to the start of the frame (see figure 2.5).

Last-Steps

In addition to their part-object-step relatives, a last-step makes sure an object or a pattern is returned while no part object is ever returned.

`out-step, up-step, and down-step` are all like their corresponding part-object-steps, but when the final part object has been found, its associated object is picked.

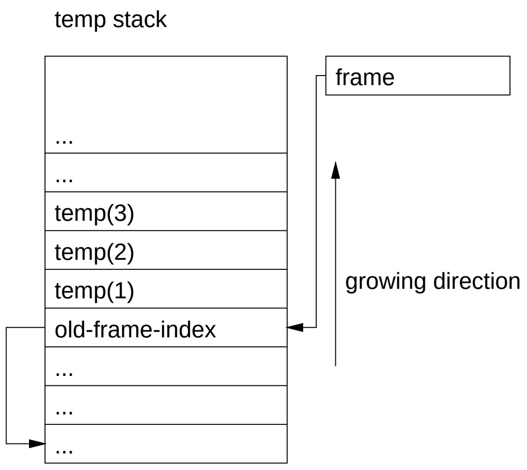


Figure 2.5.: *The temp-step picks* `temp(` *<int>* `)` *in the temp stack with the shown frame*

`lookup-step` returns the value of the attribute. This is either an object or a pattern.

`lookupIndirect-step` returns the value of the object reference or pattern reference found.

`temp-step` returns the object at <int> relative to the start of the frame (see figure 2.5).

To conclude our discussion about run time paths, we will give two explained examples of these from a real gbeta byte code file (useObserver.gbc from the nodist directory of the 0.81 distribution).

Run-Time Path Example 1

```
PUSH-ptn {<-1,~'textAndWindow.gb:280,"refresh/0"}
```

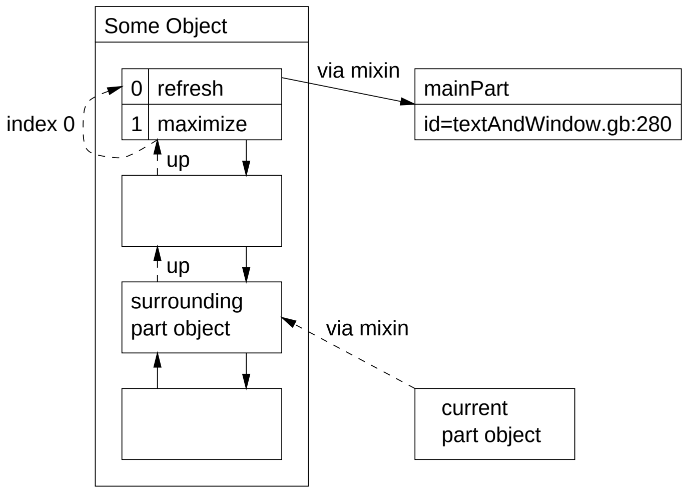


Figure 2.6.: *Evaluation sequence of the run-time path in example 1. The run-time steps are shown with dashed arrows*

The first step `<-1` is an out-step from the current part object to the surrounding part object. The next step `~'textAndWindow.gb:280` is an up-step from the part object found in the previous step. The part object with <mainPartId> , `textAndWindow.gb:280`, is found in the more general direction. The final step, `"refresh/0"`, is a lookup-step which identifies the attribute at index 0 in the part object from the previous step. Finally, the pattern is returned to the instruction (which should push the pattern onto the pattern-stack). The evaluation is shown in figure 2.6.

Run-Time Path Example 2

```
POP-objref {tmp(1),~'useObserver.gb:332,'theSubject/0'}
```

The first step, `tmp(1)`, takes the object at position 1 on the temp-stack and takes the most specific part object in that object. The second step,`~'useObserver.gb:332`, searches upwards from the part object for a part object with <mainPartId> `useObserver.gb:332`. In that part object index 0 contains an object reference which is identified with the final run-time step, `'theSubject/0'`. The instruction finally pops an object reference, type-checks it, and installs it in the found location. In the future the type check will be a separate instruction. [1]

2.2.3. Single-Line Instructions

The most common type of instruction is the single-line instruction, which is characterized by only taking one line divided into the name of the instruction and zero or more arguments. The complexity of single-line instructions varies greatly from simple `PUSH` instructions to the more complex `MERGE` instructions.

The `CALL <rtp>` instruction is an example of a single-line instruction with a run-time path as argument. It evaluates the run-time path and executes the object obtained from this evaluation.

2.2.4. Multi-Line Instructions

Multi-Line instructions are actually high level structures which eliminate the need for explicit jumps and labels. They are used both alone and nested. To give an example of a multi-line instruction we present the `generalIf` instruction. The grammar for the `generalIf` instruction is as follows:

```
        <genIf>  ::=  'generalIf(' <type> <evaluation> <alternative> * <elsePart> ')'
  <alternative>  ::=  ('|case' <evaluation> )+ '|then' <imperatives>
     <elsePart>  ::=  '|else' <imperatives>
   <evaluation>  ::=  <instruction> *
  <imperatives>  ::=  <instruction> *
  <instruction>  ::=  <singleLineInstruction> | <MultiLineInstruction>
         <type>  ::=  'bool'|'char'|'integer'|'real'|'object-reference'|
                      'pattern-reference'|'NONE'
```

The `generalIf` instruction is evaluated by evaluating the first <evaluation> and obtaining a value V as a result. Then each <alternative> is considered in the order they appear by evaluating their <evaluation> following `'|case'` and comparing their result with V. If the result is equal to V, the <imperatives> following `'|then'` are executed and the rest of the <alternative> s are ignored. If no <alternative> is chosen, the <imperatives> following `'|else'` are executed. An example of a `generalIf` instruction, taken from the file 026.gbc, again in the nodist directory, is shown in figure 2.7.

[1]Since January 2001 `CHK_QUA_OBJ` and `CHK_QUA_PTN` have been added to the set of gbeta byte codes. These instructions are inserted when the type safety of the assignment cannot be guaranteed

```
generalIf(integer
   PUSH-integer {"N/0"}
   RESETFRAME
|case
   PUSHI-integer 2
   RESETFRAME
|then
|else
   ...
   RESETFRAME
)
```

Figure 2.7.: *An example of the general if multi-line instruction*

3. Virtual Machine Architectures

A *virtual machine* is an abstract computer implemented in software [Ven96]. It executes a well-defined byte code format consisting of byte code instructions that can be compared to the machine instructions of a hardware CPU.

The great advantage of a virtual machine and byte codes is the portability achieved. Because byte codes are machine independent, they can be compiled on one platform and interpreted on another platform. But if too many different byte code formats are used by different virtual machines on the same system intercommunication between these systems may be a problem. Another benefit of the virtual machine approach is that it is easier to implement and debug a system with a compiler that compiles to byte codes executed by a virtual machine than a compiler compiling to native code.

Virtual machine architectures is a large research topic and thorough investigation and presentation of the numerous different kinds of virtual machines is out of the scope of this thesis. Instead the purpose of this chapter is to give an overview of the three most common types of architectures. Common for all virtual machines is that they execute the byte codes produced by a compiler, but different approaches can be taken. This chapter will describe three different architectures, namely instruction interpretation, just in time compilation, and dynamic compilation. Finally, the different architectures are compared with respect to ease of implementation and efficiency.

3.1. Interpreting Byte Codes

An *interpreting virtual machine* works the way depicted in figure 3.1. The source code is compiled to byte codes by a separate compiler. This is given to the virtual machine for interpretation.

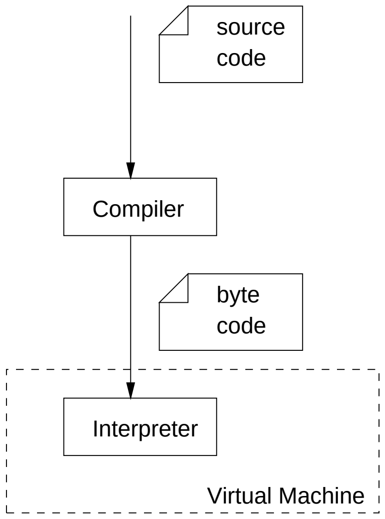


Figure 3.1.: *The data flow in a virtual machine interpreting byte codes*

In the virtual machine a stack and virtual registers can be used for storage of method parameters, local variables, intermediate results of calculations, and return values amongst other things. A heap can be used to store objects, and another area in the addressable memory can be used to store instruction byte codes – in the Java virtual machine, the area with instruction byte codes is called the method area and the next byte code to execute is pointed to by a program counter [Eng99].

Besides interpreting the byte codes, a typical virtual machine can also handle thread synchronization and garbage collection.

A byte code typically consists of a one byte opcode identifying the operation to be performed, and zero or more bytes of operands used in the operation. [Ven96, Nel79]

3.2. Just in Time Compilation

Instead of just interpreting the byte code, a virtual machine with a *just in time compiler* (JIT compiler) compiles byte codes of a method to native code the first time a method is called (see figure 3.2). The native code is cached and each time a method is called its native code is executed on the target CPU [YMP$^+$99, Sag00].

The main goal of a JIT compiler is to generate efficient native code quickly and one of the tasks connected to this is to make efficient use of the registers of the target CPU. But it is important to clarify here that the code generation must first and foremost be quick and that the efficiency goal therefore may be subordinated.

Constructing a JIT compiler is different, compared to the construction of an ordinary compiler. Compilation is done at run-time because aspects such as security can then be taken into account and especially because the portability of byte-codes is then maintained [CFM$^+$97]. Because compilation is done at run-time, compilation speed becomes crucial. Another aspect that has to be taken care of is interaction with the virtual machine. The virtual machine should for instance be able to do garbage collection even though the methods

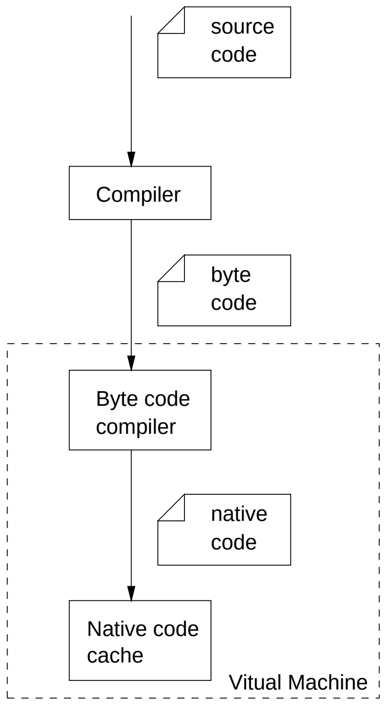


Figure 3.2.: *The data flow of a just in time compiling virtual machine*

have been compiled to native code using real registers instead of virtual registers and a stack. Thread synchronization and exception handling is also performed in interaction with the virtual machine.[CFM+97]

The performance benefit of virtual machines with JIT compilers compared to interpreting virtual machines depends on the type of application. When looking at Java virtual machines, experiments show that on average 68 % of the execution time in ordinary Java virtual machines is used interpreting the Java byte code [CFM+97, BG00]. This means that it is only about two thirds of the execution time that is optimized when using a JIT compiler. Benchmark tests modeling different types of Java applications show speedups compared to interpretation of a factor between 2.1 and 9.0 [CFM+97].

In rare cases a virtual machine with a JIT compiler can be slower than an interpreting virtual machine. If a method is only executed rarely, it may take longer to compile it and run the native code than interpreting its byte code directly. This observation is the motivation for dynamic compilation.

Instead of compiling byte codes to native code, it would be possible with gbeta byte code to compile it to a lower level byte code format. This could eliminate multi-line instructions and optimize run-time paths yielding faster execution.

3.3. Dynamic Compilation

A virtual machine with a *dynamic compiler* combines the best of interpreters and JIT compilers. It monitors the interpreter's execution of byte codes and decides which methods to compile into native code. Methods are only compiled into native code if it looks like this would make the application run faster, so methods are only compiled when it becomes evident that they are frequently used.

The architecture of a virtual machine with a dynamic compiler can be seen in figure 3.3. The profiler monitors the byte code interpreter and notifies the dynamic compiler if a method should be compiled to

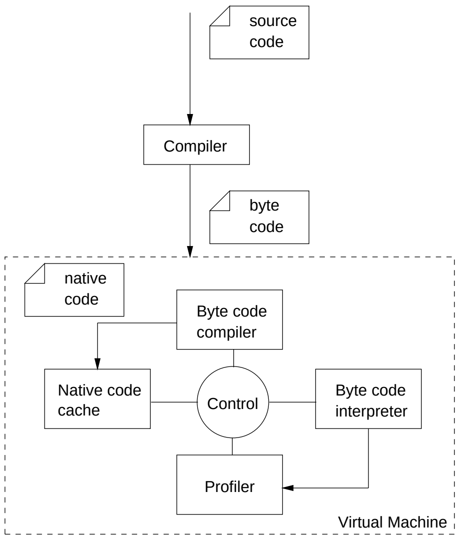


Figure 3.3.: *The components and data flow in a virtual machine with a dynamic compiler*

native code. The dynamic compiler saves the native code of a method in the native code cache to ensure that it can be used instead of the byte codes. The control module handles control of execution between the other modules.

One of the things that makes a virtual machine with a dynamic compiler smart is the following. Because the interpreter is always able to execute the byte code, the dynamic compiler can optimize the compiled code to only include a normal execution case [Arm98]. That is, things like exception handling may not be included in the native code. If an exception or other rare things occur, the virtual machine can always go back to interpreting the byte code.

The performance of dynamic compiler is generally better than both an interpreting virtual machine and a virtual machine with a JIT compiler [Arm98].

3.4. Architecture Comparison

When comparing the three architectures with respect to implementation complexity, it is obvious that instruction interpretation seems to be the easiest architecture to implement. Both just in time compilation and dynamic compilation seem somewhat more complex to implement because a compiler must be implemented, but with the profiler added to the dynamic compilation architecture, it must be the most complex to implement.

As mentioned in the previous section the dynamic compilation architecture is the most efficient closely followed by the just in time compilation architecture. Although the interpretation architecture is the simplest to implement, it is also the architecture that in general yields the least efficient execution.

It is possible to find articles that both agree and disagree with the above performance postulate. In the JIT community they find dynamic compilation slower than efficient JITs [YMP$^+$99] and in the HotSpot-dynamic compilation community they say the opposite [Arm98].

4. Memory Management

The purpose of automatic memory management is to make garbage collection possible. This means that the programmer does not have the burden of explicit memory deallocation, which can cause problems like memory leaks and dangling pointers to deallocated memory. Also, explicit deallocation is complex and error prone e.g., with complex data structures it may be difficult to place the responsibility of deallocation or to determine when an object is no longer in use.

An automatic memory management system takes care of deallocating memory which is not in use by the program any more. The problem of freeing these memory segments automatically is known as garbage collection, and several algorithms have been proposed for this task, each with its own advantages and disadvantages.

In section 4.1 we establish some terminology and discuss the important properties of garbage collection algorithms, and in section 4.2 we take a look at the most commonly used algorithms and put them in relation to these properties where appropriate.

4.1. Garbage Collection Algorithm Properties and Concepts

Common to most garbage collection algorithms (except reference-counting) is that they need to know the objects in the *root set*. The root set is the set of variables in processor registers, on the stack(s), and in global variables, all of which contain references to the memory area managed by the memory manager. This memory area is also known as the *heap-space* or just the *heap*.

When memory, previously consumed by one or more objects, is made available for future allocations or freed, we say that it is *reclaimed* or *garbage collected*.

The set of *semantically live objects* are the objects which will be used in future execution. But it is computationally undecidable to determine this set and in the context of garbage collection, the set of *live objects* is the set of objects that have been allocated which the executing program may potentially access again. This set can be found as the transitive referential closure of the root set [App98]. Intuitively this is the set of objects that can be reached by following references from the root set.

Correspondingly, the set of *semantically dead objects* are objects that will not be used in future execution and this set is of course undecidable too. Therefore in context of garbage collection, the set of *dead objects* contains objects which have been allocated but that are no more live, i.e., not reachable by following references from the root set. The really important property is that the set of dead objects is a subset of the set of semantically dead objects, because this means that we can safely reclaim the set of dead objects.

When a garbage collector *scavenges* the heap-space, it identifies live objects and makes sure these objects are not reclaimed while everything that was not live is reclaimed.

The execution of a program typically involves changes to the graph of live objects. When discussing garbage collection, we say that these changes are performed by the *mutator*.

A correct garbage collector satisfies the following properties:

Never collects live objects No live objects will ever be collected.

Collects dead objects All dead objects will be garbage collected eventually.

Maintains object graph If there is a reference in object o_i to object o_j, the mutator must always perceive the reference as such (unless it changes it itself). This should hold for all references in the object graph. That is, the work of the garbage collector should be transparent to the mutator even if they run concurrently.

The rest of this section contains some important properties of garbage collection algorithms, and advantages/disadvantages to keep in mind when studying garbage collection algorithms. It is primarily based on a talk given by Lars Bak, who is a co-inventor of the HotSpotvirtual machine at Sun Microsystems, Inc. [BG00].

4.1.1. Accurate vs. Conservative

An *accurate garbage collector* knows where in memory it has references and knows the root set of the program, so it has accurate knowledge of the object graph of live objects. An advantage of accurate garbage collection is that the garbage collector knows what variables are references, so non-pointer data like e.g., an integer cannot disguise itself as a pointer and thus keep an object artificially alive creating a memory leak. Another advantage of knowing the references is that objects can be moved. This allows compaction of the heap so heap fragmentation can be avoided. The consequence of no fragmentation is that memory allocation can usually be performed in constant time.

A *conservative garbage collector*, on the other hand, is a garbage collector that does a conservative guess at what objects are live. The reason for taking a conservative guess can be that the language in which the program is written has a liberal memory-access/type-system policy like `C/C++`. Another reason could be, that the garbage collection algorithm is trying to be smart by doing approximations in determining the live objects.

Because the conservative garbage collector must not collect any live objects, conservative garbage collection is sometimes forced to leave objects in memory that are not actually live. The reason is that conservative garbage collectors cannot tell if a word in memory seemingly pointing to an object is in fact a pointer or an integer instead and therefore the object along with everything it can reach is considered live.

Furthermore, in the case where not all references are known, objects cannot be moved, at least not when direct pointers are used, and therefore the heap cannot be compacted so the heap may suffer from fragmentation. For this reason, the memory manager will have to maintain some data structure indicating where in the heap new objects can be allocated. Allocation of memory for new objects by searching this data structure is typically slower than constant time and in addition there will in time be more small memory blocks that are useless and hard to keep track of.

In general, the conservative garbage collection approach should be avoided if the programming language allows it because of these serious disadvantages.

4.1.2. Handle Based vs. Handleless

In *handle based garbage collectors* objects must be accessed through an object handle. The idea behind handles is that when moving objects only one pointer has to be changed because other objects point to the handle instead of the object directly. Thus, heap compaction can be used, but handles use extra memory and the indirection leads to lower performance. Another disadvantage of handles is that if the data-structure containing the object handles has a fixed size, it limits the total number of objects in the heap. This was a problem with the first generations of Smalltalk implementations [CWB86].

In *handleless garbage collectors* references point directly to objects. This leads to higher performance but it is harder to move objects because all objects referencing a moved object must have their pointers updated.

4.1.3. Partial vs. Full

A *partial garbage collector* only scavenges a fragment of the heap at each garbage collection. The reason for only doing partial garbage collection is that pause times introduced by the garbage collector can be reduced this way. This is very important in applications with real time constraints or human interaction. There are other penalties to be paid with partial garbage collection, however. Often a change to a reference induces additional overhead, because the partial garbage collector has to have some extra information on which objects are referenced from where, in order to avoid scanning the entire heap during garbage collection. In addition [SaCL00] points out that generational garbage collection often has lower efficiency when the number of generations increase due to smaller collected area and write-barrier overhead.

Full garbage collectors scavenge the entire heap at each garbage collection. This can cause long pauses in program execution. The pauses are proportional to the heap size or the number of live objects depending on the chosen garbage collection algorithm. Due to their efficiency in collecting all garbage and low mutator overhead during execution, full garbage collectors may be preferred in non-interactive applications.

4.1.4. Cooperative vs. Concurrent

The execution of the program (the mutator) is stopped in a *cooperative garbage collector* implementation. The advantage of this approach is that the object graph is frozen during garbage collection but since the program execution is stopped, pauses in the execution will occur.

A *concurrent garbage collector* collects garbage without stopping the program execution. The obvious advantage of this approach is that no pauses due to garbage collection occur. But making the garbage collector and the mutator run concurrently is complex because the object graph is mutated during garbage collection and furthermore expensive synchronization is needed.

4.1.5. Single Threaded vs. Multi Threaded

A *single threaded garbage collector* collects garbage using only one thread. It is the most simple approach since no synchronization is needed between garbage collection threads. The problem with single threaded garbage collectors is that they do not scale to multi processor architectures. The reason is that the garbage collector thread can only use one processor so especially if the cooperative approach is chosen resources are wasted. Therefore the more complex *multi threaded garbage collection* approach can be used on larger systems where the garbage collection is handled by several cooperating threads.

4.2. Garbage Collection Algorithms

This section will give an overview of common types of garbage collection algorithms.

4.2.1. Reference Count

A *reference counting garbage collector* stores a reference count in each object indicating how many references are pointing to the object. If the reference count of an object is zero, the object can be garbage collected, and the objects referenced from it will, as a consequence, have their reference count decremented. To ensure that this algorithm works, the reference count must be updated each time a reference is made to or removed from an object.

An advantage of this algorithm is that it is simple. Another advantage is that it can be made incremental since this requires a check for a reference count value of 0 after decrementing. This also means that garbage can be reclaimed early i.e., there is no need to wait for the next scavenge before garbage will be reclaimed. In more real-time oriented applications a reclaim queue has to handle memory reclamation when the reference count of one object reaches zero since it could result in a lot other objects getting their reference count decremented to zero too. It may still be a problem if an object, which has a lot of references, is to be reclaimed, though, because handling this single object may take too long in a response time critical application.

A disadvantage of the algorithm is that it cannot reclaim cyclic garbage because reference counts are only a conservative approximation of liveliness. If for instance a dead object o_1 has a reference to a dead object o_2 and o_2 has a reference back to o_1 their reference counts will never reach zero. For this reason cyclic structures will have to be broken explicitly if they are to be reclaimed. Another disadvantage of this algorithm is the overhead associated with maintaining the reference counts. Finally, the interface between the mutator and the collector is complicated by the need to maintain updated reference count values – especially in parallel environments.

In practice the disadvantages of the reference count approach outweighs the advantages and therefore it is rarely used [App98, JL96, Wil92].

4.2.2. Mark and Sweep

Marking an object means setting a bit somewhere in its header, so that it is obvious to the garbage collector that this object has been processed. The idea of a basic *mark and sweep garbage collector* is that after marking objects reachable from the root set, all unmarked objects must be dead. The algorithm is divided into two phases the mark phase and the sweep phase. In the *mark phase* all reachable objects are marked using for instance a depth-first traversal from the root set. The *sweep phase* then scans the heap-space from one end to the other reclaiming all unmarked objects and unmarking all marked objects in order to make the heap ready for the next garbage collection. In the basic version this algorithm needs to interrupt the execution during the mark and sweep phases.

Advantages of this algorithm are that it is simple and easy to implement, it is capable of reclaiming cyclic structures, and there is no extra overhead associated with pointer operations during execution. Also, it can be implemented as a conservative collector which expands its area of application.

Disadvantages are that, like all non-incremental garbage collection algorithms, it can cause long pauses in the program execution, because the sweeping phase has to visit all objects in the heap, dead or live. Finally, it must manage free space in a fragmented heap, causing higher overhead for allocation, at least if handles are to be avoided. [App98, JL96]

The *mark and compact algorithm* tries to overcome the fragmented heap problem by compacting the heap in the sweep/compact phase by moving live objects to a continuous block at the beginning of the heap. Since objects are now moved, this algorithm needs to have pointer knowledge i.e., be accurate. In addition extra passes are needed to calculate the new positions of objects and update the pointers in objects accordingly, which may add significant overhead [Wil92].

4.2.3. Copy Collection

The *copy collector algorithm* works by dividing the heap into two semi spaces, the *from-space* and the *to-space*. Allocation of new objects is done in a stack based manner in the from-space. When the from-space is depleted, the garbage collection takes place. The garbage collection process begins with the root set and copies all the live objects in a breadth-first traversal to the to-space. All the live objects are then copied to the to-space, and pointers in the objects, on the stacks, and in the registers are updated accordingly. When all live objects have been copied, the rest is garbage. This is often referred to as a *Cheney scan*. Although the original Cheney scan algorithm uses a breadth-first-copying algorithm [App98] it is also possible to do a depth-first-copying instead yielding better locality of reference.

As this algorithm copies all live objects, it works most efficiently when the garbage/live objects ratio is high because the copy collector only spends time on live objects and with a high garbage/live object ratio a large heap can be collected with a low effort. One way to achieve this is not to garbage collect very often because many objects die young [LH83]. The problem with this approach is that it requires a lot of memory. Another problem with the copying algorithm is that moving large objects around in memory is expensive. A separate non-copying object space for large objects can be used for reducing this overhead. [Wil92, UJ88, App98]

4.2.4. Generational Collection

Generational garbage collectors exploit the empirical observation that in most programs there tends to be a high frequency of short lived objects [LH83]; after a given object has survived a number of garbage collections, it tends to survive for a long time. In other words the probability of death for an object decreases with its age.

The aim of generational collectors is to concentrate the collection effort on the objects that are most likely to be garbage, which, according to the above observation, are the young objects.

A *generational garbage collector* divides the heap into generations G_0, G_1,...,G_n. G_0 contains objects younger than G_1, which contains objects younger than G_2 etc. G_n is sometimes referred to as the *mature object space*. The age of an object is typically measured in how many times the object has been scavenged. Garbage collection is done more often the younger the generation is, i.e., more often in G_0 than in G_1 than in G_2 etc. The younger generations are typically smaller than the older ones and can be scavenged without scavenging the older generations. This means that generational garbage collectors do not need to scavenge the whole heap each time and pause times can thus be reduced, which is an important property in interactive systems.

The actual garbage collection algorithm in each generation can vary, but complexity increases if different algorithms are used. It is common to use a copy collector in the young generation since this algorithm is efficient when the garbage/live object ratio is high [App98].

A disadvantage of generational collection is that handling generations and especially moving objects between generations increases implementation complexity [App98].

Intergenerational References

A problem with having more than one generation is that objects from one generation can have a reference to objects in other generations implying that when garbage collecting a specific generation, it is necessary to know which objects from other generations are referencing objects in the generation to be garbage collected. One solution to this is to scan all generations each time a specific generation is to be scavenged but this is a great overhead and it also somewhat defeats the purpose of a generational garbage collector.

A common solution, to this problem, is to somehow remember references in other generations to the specific generation. A *write barrier* can be used to trap pointer modifications and use these pointers as a pseudo root set when the generation is collected.

Furthermore, it is common practice to only remember references from older to younger generations in order to save memory and time. The effect of this is that a generation cannot be scavenged without scavenging all younger generations at the same time. The positive sides to this are that it is possible to scavenge more effectively when a larger chunk of the heap is collected at once, and older generations are scavenged much less frequently. The negative side is that when such a scavenge is performed it is very likely that it will take more time than just scavenging the younger generations perhaps causing a noticeable interruption in an interactive system.

At least two ways of tracking intergenerational references have been proposed: card marking and remembered sets [Wil92, HH93]. *Card marking* divides the heap into cards of fixed size. For each such card there is a bit or byte (depending on implementation) in a card marking table that will be set unconditionally each time a reference is modified in the card. When e.g., the young generation is scavenged, all marked cards will have to be scanned for references. If scanning a card reveals no intergenerational references, the card mark can be cleared. It is important to find the right granularity for the cards; if the cards are too large, a single reference can cause significant overhead while too small cards will require too much memory for the card-marking array. Hardware pages have been suggested for implementation and rejected because stock hardware typically uses too large pages and the virtual memory system has to be modified since operating systems typically do not provide facilities for examining the dirty bits of pages [Wil94]. A study by [HMS92] concludes that the best card size on average is 256 bytes.

Remembered sets on the other hand use a more exact representation for remembering which object an object is referenced from. The advantage of this is that a lot of scanning is eliminated because either the referencing pointer is remembered directly or only the referencing object has to be scanned. The major drawback to this approach is that it can consume a lot of memory and uses more time during execution. The actual scavenge is typically faster than a scavenge using card marking which is desirable if eliminating pauses is of primary concern. Popular objects generate huge remembered sets and may require special handling such as keeping the object in a place where it is not moved.

Hybrid approaches have been implemented in [HH93, Hud00] to combine the precision of remembered sets with the run-time efficiency of card marking. At run-time card marking is used but before each scavenge the marked cards are summarized into remembered sets used as basis for the scavenge. A more dynamic hybrid able to switch between pure card marking and card marking combined with remembered sets is also suggested in [HH93] to yield better performance.

4.2.5. The Train Algorithm

The *train algorithm* is an example of an incremental garbage collection algorithm. Incremental garbage collectors try to minimize the pauses introduced by garbage collection. The idea is to only garbage collect a bounded part of the heap each time the garbage collector is run. This makes the garbage collection algorithms more complex but systems with incremental garbage collection provide more usable applications in many cases.

The basic idea of the train algorithm is to cluster related referencing objects into the same cars or trains (see figure 4.1) and check if there are any references to a specific car or train. If there are not, the objects in the car or train can be garbage collected, and the whole car or train can be reclaimed.

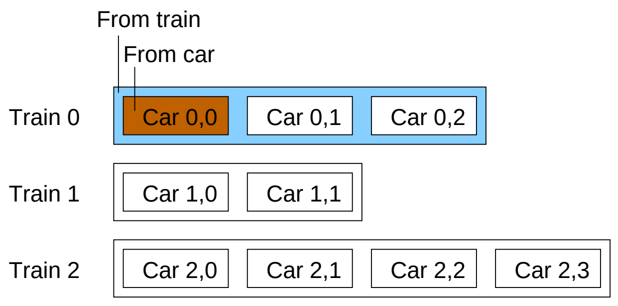


Figure 4.1.: *The organization of the heap using the train algorithm*

To maintain a total ordering of cars each train and car is given a number (see figure 4.1). The oldest of the trains i.e., the train with the lowest number is called the *from train*. The *from car* is the oldest car in the from train i.e., the car with the lowest number in the from train.

The train algorithm, as defined as a mature generation collector in [GS93], is presented below:

1. First, check to see whether there are external references into the from train. (This is done by inspect-

ing the train roots and the remembered set associated with the train being collected.) If this is not the case, then free the entire from train.

2. Otherwise, start cleaning up the from car as follows:

   a) Move objects referenced from other trains (as found in the from car's remembered set) to those trains, and move objects referenced from outside mature object space to any train except the from train, perhaps an entirely new one.

   b) Also, move objects being promoted from younger generations to mature object space into any train, except the from train.

3. Then evacuate the followers (i.e., objects referenced directly or indirectly by the moved objects) in the from car by scanning over the objects moved in the previous step and evacuating, in typical copy collector style, all reachable objects to the trains from which they are now referenced.

4. At this point, the from car may still contain objects referenced from the outside (namely those that are only reachable from other cars of the from train). Move these objects into the last car of the from train, appending a new car should the train run full. Then free the space used for holding the from car.

In [GS93] certain points are further clarified. The most important is that to increase locality of reference when moving objects referenced from other trains and their followers the algorithm must try to move them to the car from where they were referenced or else the train from where they were referenced. It is also proposed that to increase locality of reference, the phase where objects referenced from other trains (2.a) are moved could be intertwined with evacuating the followers (3). The reason is that the probability of the destination car being able to hold the followers of an object moved in phase (2.a) is increased.

How the Train Algorithm Works

To explain the train algorithm in more detail a little example is presented. To make the example simple, cars can only contain three objects.

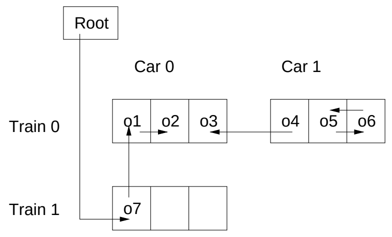


Figure 4.2.: *The initial situation*

The initial situation can be seen in figure 4.2. There are three live objects, namely the objects o_1, o_2, and o_7. The rest of the objects are dead and therefore the train algorithm garbage collector will collect them.

During the first garbage collection both the from train and from car are referenced from outside and therefore the from car is scavenged. First, objects from other trains are moved to the car from where they are referenced, in this case object o_1 is moved to car 0 in train 1. Then followers of object o_1 are evacuated and therefore object o_2 is moved to car 0 in train 1. Finally, objects referenced from within the from train are moved to the end of the from train allocating a new car if the last car in from train is full. This results in the addition of a new car (car 2) to the from train and the movement of object o_3 to this car. The resulting heap can be seen in figure 4.3.

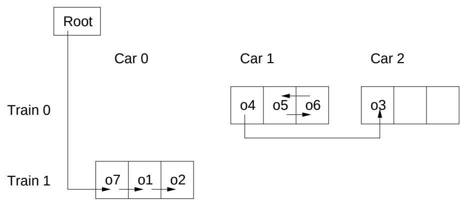


Figure 4.3.: *The heap after the first garbage collection*

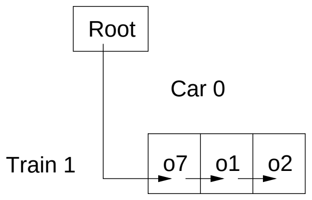


Figure 4.4.: *The heap after the second garbage collection*

The second garbage collection results in the from train being collected because it has no external references (see figure 4.4).

The example gives insight into how the algorithm works, but some argumentation is needed to clarify that the algorithm will collect large cyclic garbage structures. It is easy to see that if a large garbage structure is contained entirely within a train it will be collected. As the train is processed, all objects referenced from other trains or from the root will be evacuated. Garbage will either be directly collected or moved to the end of the train and eventually all externally referenced objects and their followers will have been moved out of the train leading to the whole train being collected. So if it is the case that large garbage structures always end up entirely within one train they will be garbage collected. This is the case since we only move objects to trains holding references to them and eventually a garbage structure spanning several trains will end up in the highest numbered train of which it was initially part.

The advantages of the train algorithm are that it is incremental i.e., non-disruptive, it generates a high locality of reference i.e., objects referencing each other are placed physically close. A disadvantage of the algorithm is that it seems more complex to implement than the other mentioned algorithms.

5. The Virtual Machine

In [IP01] we documented the design and implementation of a prototype virtual machine able to execute a limited number of gbeta byte code instructions. The main achievement of this work was an implementation of the gbeta entities, such as patterns, mixins and part objects, which were necessary to execute gbeta programs. In the evaluation of this implementation we identified a number of weak points which prevented this virtual machine from using our memory management component.

The main problem with this implementation was that our execution model prevented us from having well defined safe points. A *safe point* is a point during the execution of the program where it is safe to do copying collection, i.e., after copying garbage collection at a safe point, the virtual machine will be able to continue executing without problems. The used execution model, called the `.execute`-model, used `execute()` methods on `Objects`, `PartObjects` and `Instructions`. A garbage collection during execution using this model could imply that the `C++` run-time system got invalid this-pointers on its execution stack because for instance invoked part objects could have been moved due to garbage collection. We previously referred to this problem as the *this-problem*.

In [IP01] we proposed that using a more flat execution model could solve the problem. In the *flat execution model* a loop continues to execute instructions (one at a time) using a switch statement to decode the next instruction to be executed. In essence this model ensures that no fragile data, e.g., this-pointers, are left on the `C++` execution stack between the execution of instructions. The idea behind this model is further explained in section 5.2.

This chapter documents the design and implementation of a gbeta virtual machine using a flat execution model able to execute a majority of the existing gbeta byte code instructions. But first we recapitulate the architecture of this virtual machine by reviewing some slightly changed versions of the class diagrams in [IP01] (a detailed design document, describing the design of the first virtual machine, is present in chapter 6 of [IP01]). As can be seen in figure 5.1, the main class diagram has not changed much. The main change is that the `Thread` class has got two new stacks and a program counter (`programCounter`) attribute. This attribute points to the next instruction to be executed or, if the `Thread` is currently executing an instruction, it points to the instruction being executed. It is the responsibility of the instruction to change the program counter, which makes it easy to make jumps (and infinite loops if one forgets to update the program counter).

Also, the relations between the gbeta entities, used by the virtual machine, has not changed much as can be seen in figure 2.1. The main change in these classes is that the `execute()` and `initAttributes()` methods of `Object` and `PartObject` have been removed, and methods to get the first instruction, used when executing or initializing, have been added instead.

The purpose of this virtual machine is to enable experimentation with our memory management component. Therefore we have decided to implement enough instructions to allow execution of simple gbeta programs. This implied that we omitted instructions supporting repetitions and concurrency. Besides that the rather complex `POP-ptn,_SPECIALIZE-obj`, which includes an operation similar to the `become:` message in Smalltalk [SUH86], is not supported; it may easily be avoided by not using certain pattern reference assignments like `x1##->x2##` in the gbeta programs being executed.

The source of the virtual machine component is placed in the `vm` directory of the CD-ROM. A few discrepancies exists between the this documentation and the actual source code. One of them is that the `ByteCodeLoader` class is called `InstructionParser` in the source code.

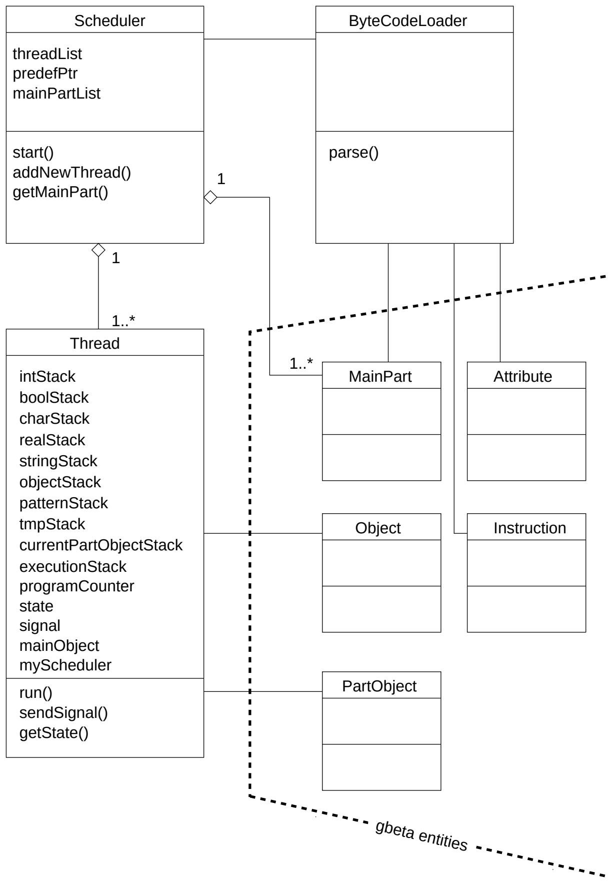


Figure 5.1.: *The main class diagram of the virtual machine. The classes* `MainPart`, `Object`, `PartObject`, `Instruction`, *and* `Attribute` *is included to show the connections between the main class diagram and the gbeta entity class diagram (see figure 2.1)*

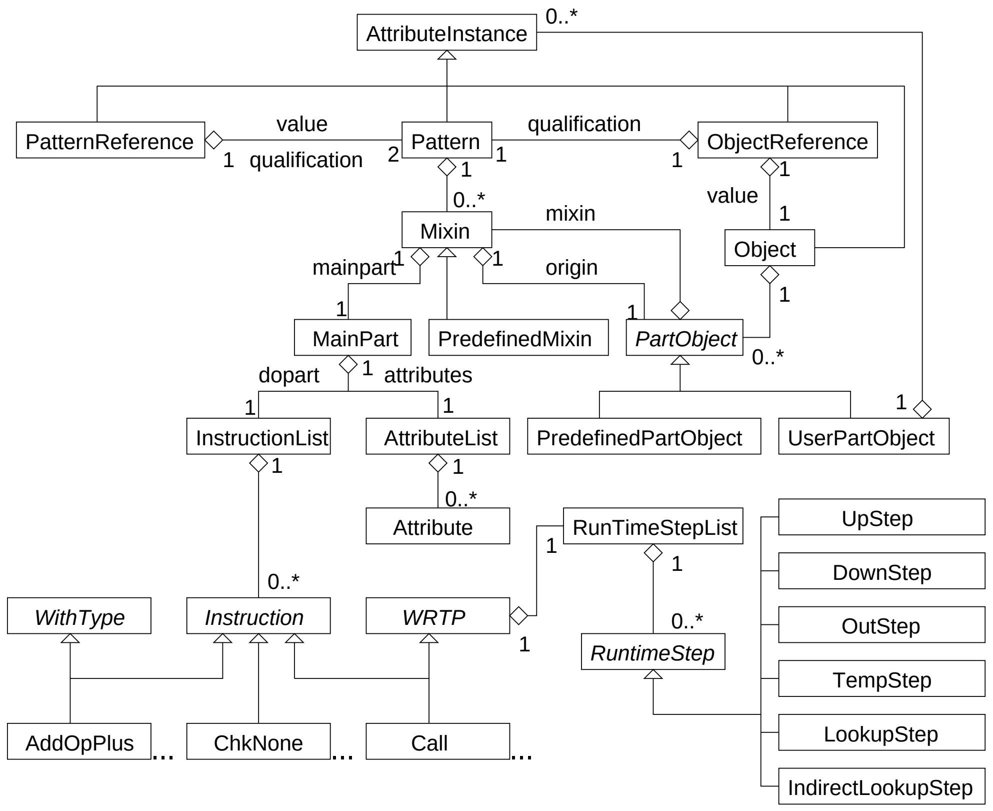


Figure 5.2.: *The class diagram of the gbeta entities used by the virtual machine. Three examples of instruction classes are presented in the lower left corner. Instruction subclasses can inherit from zero or more of the super classes* `WRTP` *and* `WithType` *which holds run time paths and type information functionality respectively. “...” indicates that there are several instructions of the given type*

5.1. Thread - Scheduler Interface

The `Scheduler` - `Thread` interface is ”two-sided”, i.e., the `Scheduler` invokes methods on `Thread` and `Thread` invokes a method on `Scheduler`. This is because we have made the class `Scheduler` have two purposes:

1. Schedule and signal `Thread(s)` to obtain safe points.
2. Support `Thread(s)` in:
   - Getting a mainpart given a mainpart id.
   - Holding a pointer to the outer-most part object (called predef) part object to ensure that it will not get garbage collected.

In figure 5.4 a sequence diagram explains how the `Scheduler` schedules and signals `Thread(s)`, and figure 5.3 illustrates how `Thread` uses `Scheduler` to lookup `MainPart` instances.

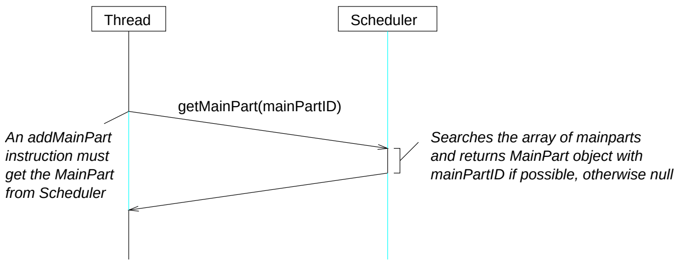

Thread
Scheduler
getMainPart(mainPartID)
An addMainPart instruction must get the MainPart from Scheduler
Searches the array of mainparts and returns MainPart object with mainPartID if possible, otherwise null

Figure 5.3.: *`Thread` - `Scheduler` interaction*

In figure 5.3 the `Thread` is executing an `AddMainPart` instruction with a main-part id as one of its arguments. To add a mixin, referring to the wanted main-part, to the pattern on top of the pattern stack, the `Thread` invokes `getMainPart(mainPartId)` on its `Scheduler` to get a reference to the this main-part.

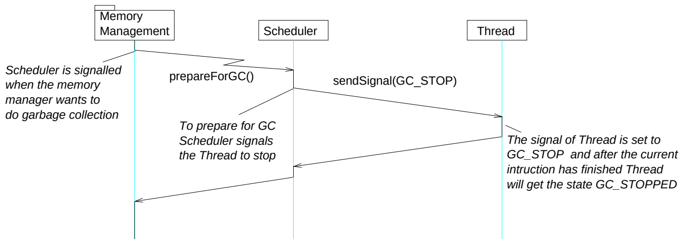

Memory Management
Scheduler
Thread
Scheduler is signalled when the memory manager wants to do garbage collection
prepareForGC()
sendSignal(GC_STOP)
To prepare for GC Scheduler signals the Thread to stop
The signal of Thread is set to GC_STOP and after the current intruction has finished Thread will get the state GC_STOPPED

Figure 5.4.: *`Scheduler` - `Thread` interaction. The memory management component is added to illustrate that the `Scheduler` must be signaled, when a garbage collection is necessary when introductory space is filled*

When a safe-point is needed the memory management component signals the `Scheduler` using `prepareForGC()`. When this method is invoked on the `Scheduler` it sends a `GC_STOP` signal to all its threads (in our case only one) using `sendSignal(signal)`. Between each instruction the `Thread` processes signals. It reacts to a `GC_STOP` signal by changing its state to `GC_STOPPED`. Besides `prepareForGC()` the `Scheduler` invokes `run()` to start the `Thread`, and restart it after a garbage collection. To restart the `Thread` the `Scheduler` also sends a `RESUME` signal, using `sendSignal()`, to the `Thread`.

5.2. Obtaining Safe Points

A set of well defined safe points is required during execution to make copying garbage collection possible. It is only safe to do copying garbage collection in our context when it can be guaranteed that the `C++` run-time systems has no pointers to objects which might get moved during garbage collection.

An *i-fetch loop* is the classic loop used in an interpreter or CPU to execute instructions. The i-fetch loop fetches the next instruction to be executed, decodes this instruction, and finally it executes the instruction before it continues to fetch the next instruction. The three main components used to obtain safe points are the execution stack, the current part object stack, and the i-fetch loop.

5.2.1. The Execution Stack

Execution happens in context of a part object. `CALL` instructions, `INNER` instructions, and similar instructions will cause execution to happen in context of a different part object. We refer to this as a *part object switch*. Moreover, execution can be divided into ordinary execution and initialization execution. Part object switches can also occur during initialization execution.

The execution stack contains pointers to instructions. The stack elements each point to the first instructions to be executed after returning from a part object switch (for instance due to the execution of a `CALL` instruction).

5.2.2. The Current Part Object Stack

The current part object stack contains pointers to `PartObject` instances. These pointers point to part objects which were previously the current part object, i.e., when an instruction switches part object, it pushes the new current part object to the current part object stack.

5.2.3. The I-fetch Loop

Instead of using `execute()` methods to execute `Objects` with `PartObjects`, we have implemented methods (called `getFirstInstruction()`) on these classes which return a pointer to the first instruction to be executed in a part object. An i-fetch loop in the `Thread` class uses these ’getInstruction’ methods to switch context when for instance a `CALL` instruction is executed. The i-fetch loop in `Thread` works as sketched in figure 5.5:

```
while(state == RUNNING){

  switch(programCounter->id) {
    case ADD-MAINPART-ID:
      //code for 'ADD-mainpart gbeta' byte code instruction
    case ADDOP-PLUS-ID:
      //code for 'ADD(_+_)' gbeta byte code instruction
    .
    .
    .
    case WHILE-ID:
      //code for 'while' gbeta byte code multi-line instruction
    default:
      print "could not decode instruction - exiting"
      exit
  }

  check signals

}
```

Figure 5.5.: *The I-fetch loop in `Thread`*

The first thing the thread does, in the i-fetch loop, is to decode the instruction pointed at by the program counter. Then the code for this instruction is executed which will set the program counter to the next instruction to be executed. Finally, the thread checks to see if it has received any signals. If this is the case it changes its state to a state corresponding to the signal.

The `Thread` is signaled by the `Scheduler` when for instance all threads must be stopped due to garbage collection. The `Thread` - `Scheduler` interface is described in 5.1.

To illustrate how we avoid using the `execute()` methods on objects and part objects, we present an example of what is done when a `CALL` instruction is executed and also when execution returns. To enable returning from part object switching instructions we made the `ByteCodeLoader` add special `RETURN` instructions to instruction lists (see subsection 5.3). This example is illustrated in figure 5.6. In figure 5.7 the pseudo code for a `CALL` instruction is presented.

```
case CALL-ID:
  evaluate run-time path to get object o to execute
  get first instruction inst to execute on o
  get first part object p to be executed in o
  push p to current part object stack
  add new frame to temp stack
  push next instruction to execution stack
  set program counter to inst
```

Figure 5.7.: *Pseudo code for a `CALL` instruction*

First, the `CALL` instruction uses its run-time path to find the object to execute. It uses this object to get the first instruction to be executed in this object. This is the first instruction in the do-part of the main-part of the most general part object in this object. This part object is then pushed to the current part object stack and a new frame is added to the temp stack (for information about the frames and the temp stack see

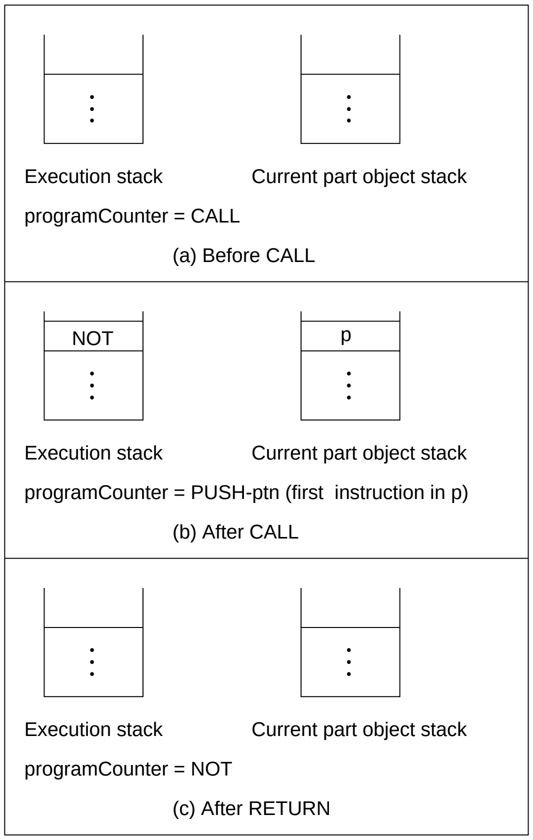


Figure 5.6.: *Illustration of what happens when a `CALL` instruction is executed and its corresponding `RETURN` instruction is executed. In the example the instruction after the `CALL` instruction is a `NOT` instruction and the first instruction in part object to be executed is a `PUSH-ptn` instruction. (a) State of the thread before the CALL instruction. (b) State of the thread immediately after the `CALL` instruction. (c) State of the thread after the `RETURN` instruction*

[IP01, page 37]). Finally, the instruction after the `CALL` instruction is saved by pushing it to the execution stack. To handle that the most general part-object in an object can be a `PredefinedPartObject`, all `PredefinedPartObjects` of a given type have a shared main-part with a do-part having a `RETURN` instruction as the only instruction.

To return from execution in the called object the `RETURN` instruction pops an instruction pointer from the execution stack. If this instruction is not null the `RETURN` instruction deletes the current frame on the frame stack, pops the current part object of the current part object stack, and sets the program counter to point to this instruction. If the instruction popped from the execution stack is null, the state of the thread is set to finished. Adding a special `HALT` instruction to the end of the do-part of the outer most main-part, would mean that the time used by the check for null in `RETURN` could be avoided.

Part object switches do not only occur when a new part object has to be executed, it also occurs when the attributes of a part object must be initialized. Attribute initialization is explained in section 5.4, but before that we introduce a number of special instructions used to implement the flat execution model.

5.3. New Special Instructions

To allow safe points within multi-line instructions and, we have defined some simple control flow instructions to which the multi-line instructions are translated.

`Return` Pops the current part object, sets the previous frame on the temp stack as the current frame, and sets the program counter to whatever is popped from the execution stack. If program counter is null after this, the instruction sets the state of the thread to `FINISHED`.

`ReturnNPPO` Sets the program counter to whatever is popped from the execution stack. No check is made to ensure this is not null. Both the temp stack and the current part object stack are unaffected by this instruction. (`NPPO` is shorthand for No *Pop* Part Object)

`ReturnMPAttrInit` A specialized instruction used when initializing the attributes of an object. The instruction first deletes the current frame on the temp stack and pops the current part object of the part object stack.

If there is a more specific part object than the current part object in the object, this part object will be pushed on the current part object stack, a new frame will be added to the temp stack, and the program counter will be set to point to the first instruction of the attribute initialization instruction list of the new current part object.

If, on the other hand, there were no more-specific part object, the instruction will return by popping the execution stack to the program counter.

Again, this instruction makes no check to ensure the program counter is not null. This is safe, since the `ByteCodeLoader` ensures that there always will be at least a `RETURN` instruction after an attribute initialization.

`ContGatherVirt` This is also a very specialized instruction. Its purpose is to make it possible to initialize virtual attributes correctly without having to do it in one step. In order to make this possible, this instruction contains information about the attribute name and index of the introducing binding as well as the main-part id.

When executed, this instruction will search the more specific part objects for a part object that has a virtual attribute matching the introducing binding's attribute name, index, and main-part id.

If such a part object is found, the current part object stack is popped and the part object is pushed, the temp stack gets its frame deleted and a new one added, and finally the program counter is set to the first instruction of the gather virtual instruction list.

If no part object could be found, this marks the end of a garther-virt instruction in the byte code. The only thing needed is to pop the current part object stack, delete the frame on the temp stack and set the program counter to the instruction after the gather-virt instruction by assigning the value popped from the execution stack to the program counter.

For further explanation of this instruction see subsection 5.4.2.

`JumpNPPO` This instruction contains a pointer to an instruction that will be executed after this one. (Note that in this case `NPPO` is shorthand for No *Push* Part Object).

`JumpTrueNPPO` is a conditional jump instruction. It pops the boolean stack. If the value is true, it jumps to the instruction pointed to in its instruction pointer attribute, otherwise the next instruction is executed. (Again, `NPPO` is shorthand for No *Push* Part Object).

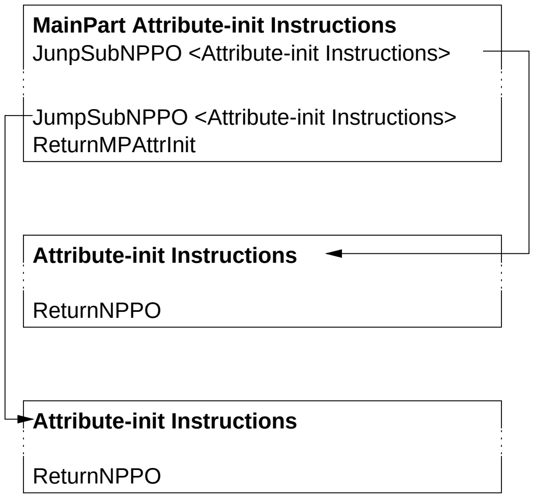
`JumpSubNPPO` is an unconditional jump to a sub routine. This means that the following instruction will be pushed onto the execution stack and the program counter will be set to the instruction pointed to by instruction pointer attribute. (Again, `NPPO` is shorthand for No *Push* Part Object).

`CopyTop` Copies the element on the top of a stack. The actual stack is specified as an attribute of this instruction.

`CopyTop2` Like the `CopyTop` instruction except that it operates on two elements instead of one. For instance if the top of the integer stack is $\{1,2\}$ before a `CopyTop2` instruction, the top of the integer stack after the `CopyTop2` instruction will be $\{1,2,1,2\}$.

5.4. Attribute Initialization

When a `NEW,_ptn->obj` instruction is executed, the `Pattern` on top of the pattern stack is popped and instantiated to an `Object`. But besides that the attributes of all part objects in this object must be initialized. The first implementation of this virtual machine simply called a `initPartObjects()` method on the instantiated object and this could result in numerous part object switches and therefore no safe points could be defined during attribute initialization [IP01]. Since it is possible to write a gbeta program where a large part or even the whole program execution takes place in attribute initialization, safe points must be present during attribute initialization.



MainPart Attribute-init Instructions
JunpSubNPPO <Attribute-init Instructions>
JumpSubNPPO <Attribute-init Instructions>
ReturnMPAttrInit
Attribute-init Instructions
ReturnNPPO
Attribute-init Instructions
ReturnNPPO

Figure 5.8.: *How the attribute initialization list of a main-part works*

To make it possible to do garbage collection during attribute initialization we made the `ByteCodeLoader` generate a special list of attribute initialization instructions connected with each main-part. Figure 5.8 illustrates how these main-part attribute initialization lists work. When an object must be initialized two things are done. First the part object to be initialized is identified and pushed to the current part object stack. This is the most general part object of the object to be initialized. Secondly the initialization instruction to be executed is identified and the program counter is set to this instruction. This is the first instruction in the attribute initialization list of the main-part of the identified part object. If the part object has any attributes the identified instruction is a `JumpSubNPPO` otherwise it is a `ReturnMPAttrInit`. The `JumpSubNPPO` instruction's duty is to push the next instruction and to make the program counter point to the first initialization instruction of one of the attributes of the main part. So, after a `JumpSubNPPO` instruction one of the attributes in the part object will be initialized. These `Attribute-init Instructions` lists are terminated with a `ReturnNPPO` instruction, which sets the program counter to point to the instruction on top of the execution stack, e.g., the next `JumpSubNPPO` in the `MainPart Attribute-init Instructions` list. The last instruction in a main-part attribute initialization list is a `ReturnMPAttr-Init`. The purpose of this instruction is to make the thread switch to initialization of the next most general part object if present and otherwise return. This way all the attributes in all the part objects of the object are initialized while creating a safe point between each instruction.

So the job of the `NEW,_ptn->obj` instruction is only to instantiate the pattern to an object *o* and make a context switch to the first instruction in the `MainPart Attribute-init Instructions` list (as described above) of the most general part object in *o*.

This form of attribute initialization is characterized as eager initialization but running gbeta programs also requires lazy initialization [BC95].

5.4.1. Lazy Attribute Initialization

The gbeta code in figure 5.9 illustrates why lazy attribute initialization is required. The value of _b is only known at run-time and its value determines which of the objects a and b in q that must be initialized first. If _b is true the initialization order must be a first then b but the opposite initialization order is required if _b is false. So, sometimes it is necessary to stop the execution of an instruction because a required attribute is uninitialized and initialize the attribute and restart the given instruction. But this requires that instructions which could result in lazy attribute initialization must be able to be restarted without side effects. This is quite simple to ensure since the evaluation of run-time paths is the only thing that can result in lazy initialization and these evaluations have no side effects.

So, if the evaluation of run-time paths is placed in the beginning of the execution code connected with these instructions and subsequently a check that initializes lazy initialization if necessary is placed, then we can stop these instructions, initialize the attributes, and restart them.

```
-- betaenv:descriptor --
(#
   _b : @boolean;
   x: @(# self: @this(object) #);
   _: (# o: ^x do INNER exit this(_)[] #);
   q: (# a: @(_(# do (if _b then x[]->o[] else b[]->o[] if)#)).o;
         b: @(_(# do (if _b then a[]->o[] else x[]->o[] if)#)).o
      #)
do
   q; true->_b; q
#)
```

Figure 5.9.: *Example of gbeta program requiring lazy initialization*

In figure 5.10 is a piece of pseudo code that illustrates what must be done after evaluation of a run-time path if lazy initialization is required.

```
if(resultOfRuntimeEvaluation == NULL){
  push current instruction to execution stack
  push RETURN instruction to execution stack
  push part object to be initialized to current part object stack
  set programCounter to first initialization instruction of attribute
  add frame to temp stack
  skip actual execution of this instruction
}
```

Figure 5.10.: *What must be done if lazy initialization is required*

The reason why a `RETURN` instruction is also pushed to the execution stack is that at the end of the attribute initialization instruction list is a `ReturnNPPO` (see figure 5.8) but a "real" `Return` is required to restart the stopped instruction. This is because the current part object and current frame must be restored after the lazy initialization has finished.

5.4.2. Virtual Attributes and Gather-virt

```
(# p: (# v:< object #);
   q: p(# v::< integer #);
   r: q(# v:: integer #);
   do r;
#)
```

Figure 5.11.: *Code example of virtual chain*

In gbeta it is possible to specialize some pattern attributes of a pattern in a subpattern. This is done using virtual declarations and virtual chains. The concept *virtual declaration* covers a virtual pattern declaration, a virtual further-binding, and a virtual final-binding. A *virtual pattern declaration* introduces a pattern attribute that can be specialized in subpatterns and is denoted with the syntax `:<`. A *virtual further-binding* specializes the introducing binding or the previous further-binding of the pattern attribute in a subpattern. Zero or more further-bindings can be present and they are denoted by the syntax `::<`. The *virtual final-binding* specializes the pattern attribute as the further-binding but it also specifies that this attribute cannot be further specialized. A virtual final-binding is denoted by the syntax `::`. So a virtual chain consists of exactly one virtual pattern declaration, zero or more virtual further-bindings and finally an optional virtual final-binding. Figure 5.12 along with the gbeta code in figure 5.11 illustrates an example of the pattern *p* with the virtual pattern attribute *v* which is the virtual pattern declaration. The pattern *p* is specialized by the pattern *q* which has a virtual further-binding of *v*. The pattern *r* specializes the pattern *q* and makes a virtual final-binding of the attribute *v*.

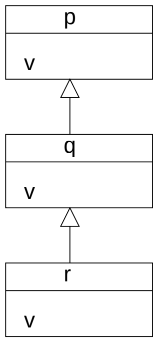


Figure 5.12.: *Example of a virtual chain*

With respect to types a virtual pattern declaration, a virtual further-binding and a virtual final binding only gives an upper bound to the type.

Having virtual attributes makes attribute initialization more complicated. The initialization of a virtual attribute requires that all part objects in an object having contributions to this attribute is used to initialize the virtual attribute.

The above program compiled to gbeta byte code is shown below. This byte code has been annotated with numbers indicating in which order the instructions are executed. The `END_OF_GATHERVIRT_CODE` delimiter can be thought of as a special instruction.

```
MainPart("‘216"
   "p/0": (
      PUSH-ptn_"object"              1
      ADD-mainpart ‘42 origin {}     2
      INSTALL-ptn 0                  3
   )
   "q/1": (
      PUSH-ptn {"p/0"}               4
      ADD-mainpart ‘108 origin {}    5
      INSTALL-ptn 1                  6
   )
   "r/2": (
      PUSH-ptn {"q/1"}               7
      ADD-mainpart ‘172 origin {}    8
      INSTALL-ptn 2                  9
   )
|
   PUSH-ptn {"r/2"}                 10
   NEW,_ptn->tmp 1                  11
   CALL {tmp(1)}                    26
   RESETFRAME                       27
)
MainPart("‘42"
   "v/0": (virtual
      PUSH-ptn {<-2,"object/0"}     13
      END_OF_GATHERVIRT_CODE        14
      GATHER-virt "v/0" in "‘42"    12
      INSTALL-ptn 0                 21
   )
|
)
MainPart("‘108"
   "v/0": (virtual "v/0" in "‘42"
      PUSH-ptn {<-2,"integer/3"}    15
      MERGE-ptn                     16
      END_OF_GATHERVIRT_CODE        17
      PUSH-ptn {^‘42,"v/0"}         22
      INSTALL-ptn 0                 23
   )
|
)
MainPart("‘172"
   "v/0": (virtual "v/0" in "‘42"
      PUSH-ptn {<-2,"integer/3"}    18
      MERGE-ptn                     19
      END_OF_GATHERVIRT_CODE        20
      PUSH-ptn {^‘42,"v/0"}         24
      INSTALL-ptn 0                 25
   )
|
)
```

Figure 5.13.: *Example 5.11 compiled to byte codes. The byte codes have been annotated with numbers indicating the order of execution*

Figure 5.13 explains how `GATHER-virt` and virtual attribute initialization works in general. When a virtual pattern declaration attribute must be instantiated the `GATHER-virt` instruction belonging to the main-part of the most general mixin will construct a correct pattern.

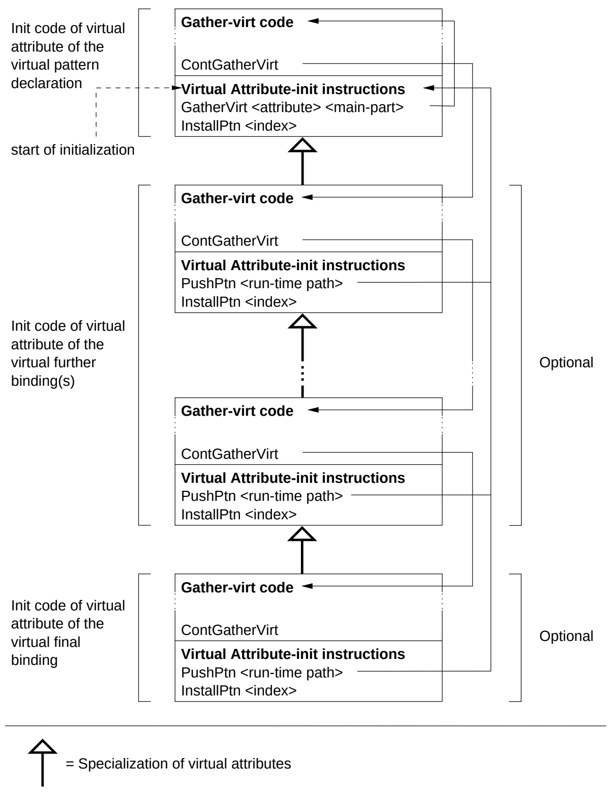

Init code of virtual attribute of the virtual pattern declaration
start of initialization
Gather-virt code
ContGatherVirt
Virtual Attribute-init instructions
GatherVirt <attribute> <main-part>
InstallPtn <index>
Gather-virt code
ContGatherVirt
Virtual Attribute-init instructions
PushPtn <run-time path>
InstallPtn <index>
Init code of virtual attribute of the virtual further binding(s)
Optional
Gather-virt code
ContGatherVirt
Virtual Attribute-init instructions
PushPtn <run-time path>
InstallPtn <index>
Init code of virtual attribute of the virtual final binding
Gather-virt code
ContGatherVirt
Virtual Attribute-init instructions
PushPtn <run-time path>
InstallPtn <index>
Optional
= Specialization of virtual attributes

Figure 5.14.: *`GATHER-virt` and virtual attribute initialization. Depending on the structure of the virtual chain, a hierarchy of one virtual attribute used for the virtual pattern declaration, zero or more virtual attributes used for futher bindings, and zero or one virtual attributes used for the final binding is present*

The initialization code of a virtual attribute is divided into two parts, the `Gather-virt Code` part and the `Virtual Attribute-init instructions` part. If the virtual attribute belongs to a virtual pattern declaration, i.e., the introducing binding, the `Virtual Attribute-init instructions` part begins with a `GATHER-virt` instruction. The functionality of this instruction is to execute the 'gathervirt' instructions and thereby create a correct pattern for the virtual pattern declaration attribute. So, if the current virtual chain consists of an introducing binding, a futher binding and a final binding, the `GATHER-virt` instruction must merge all the virtual pattern attributes with the same id starting from the most general part object.

To make virtual attribute initialization work with the flat execution model, we had to add a special instruction to the `Gather-virt Code` lists - namely the `ContGatherVirt` instruction. The purpose of this instruction is to jump to the initialization of the virtual attribute with the correct id of a more specific part object if present. This instruction is further explained subsection 5.3.

5.5. Multi-Line Instructions

This section explains the implemented multi-line instructions. To refresh your understanding of the special instructions used to allow safe points within multi-line instructions see section 5.3.

5.5.1. Named For

A named for loop in gbeta is executed with an index variable value ranging from 1 to N. The instructions of a named for are divided into two parts; the evaluation instructions giving the N and the body instructions. When parsing, these instructions are inserted into two separate lists and when the parsing of the named for instruction is finished, several instructions are appended to the two lists as shown in figure 5.15. The instructions added to the evaluation instructions can be further logically divided into initialization- and cleanup-instructions. The instructions added to the body increment the counter and checks for whether the body instructions should be executed again or the cleanup code should be executed.

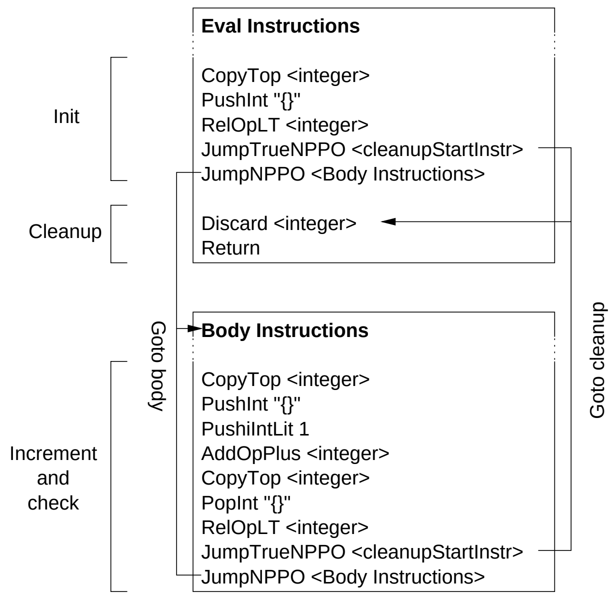


Figure 5.15.: *The instructions used for making a named for instruction*

The named for instruction itself sets up a `PrededfinedPartObject` of integer type for keeping the index variable and assigns it the value 1. This po is pushed to the current part object stack, a new frame is added to the temp stack, the instruction following the named for instruction is pushed to the execution stack, and finally the program counter is set to point to the first instruction in the evaluation instruction list.

The original evaluation instructions are supposed to leave the value N on the integer stack indicating how many times the body instructions are to be executed. This value is kept on the stack while the loop is active, and removed in the cleanup instructions. Since the `RelOpLT` instruction uses a destructive read, the `CopyTop` instruction is needed to preserve the old value. The `PushInt` {} instruction simply copies the value from the current (integer) part object to the integer stack.

5.5.2. Simple For

The simple for instruction works much like the named for version except that no part object is needed to store the index variable, and the index variable is inaccessible from the body code. The index variable is in our implementation stored on the integer stack like N in the named for instruction. The only problem this gives us is, that we need yet another instruction to be able to access both of these variables without deleting them. The `CopyTop2` instruction was invented for this purpose. The instructions inserted during the parsing can be seen in figure 5.16.

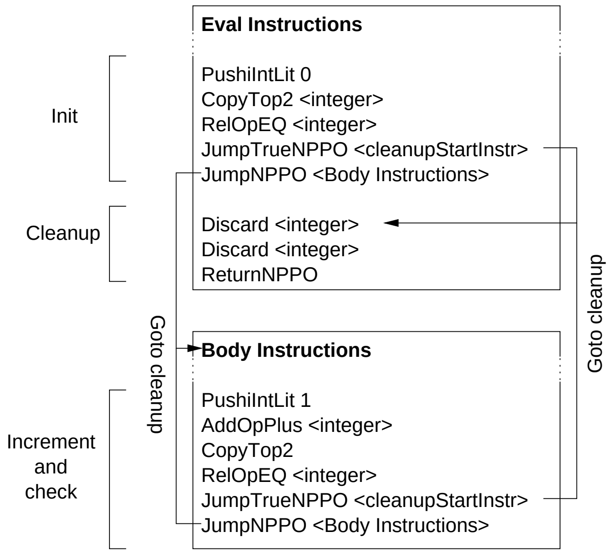


Figure 5.16.: *The instructions used for making a simple for instruction*

The instruction itself only pushes the instruction following the program counter onto the execution stack and sets the program counter to the first instruction in the evaluation instructions.

5.5.3. Simple If

A simple if instruction contains three lists of instructions: an evaluation list, a then list, and an else list. When the evaluation list of instructions has been evaluated a boolean value will be left on top of the boolean stack. If this value is true, the then list of instructions must be executed, otherwise the else list of instructions must be executed.

When the simple if instruction has been parsed, the instructions are added to the list as shown in figure 5.17.

The simple if instruction itself only transfers control to the evaluation list, i.e., it conducts a jump and saves the next execution to be executed after the simple if instruction on the execution stack.

5.5.4. General If

Because a general if construct is more complicated than the other multi-line instructions (in terms of syntax at least), we present the syntax below.

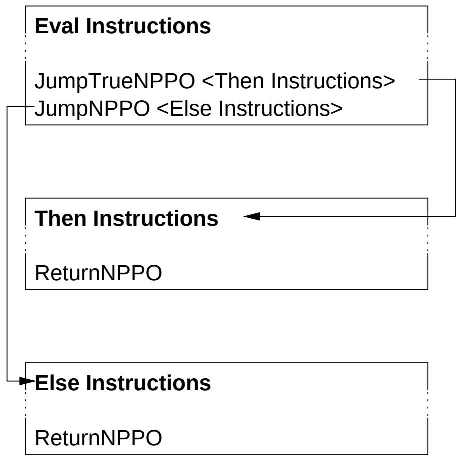


Figure 5.17.: *The instructions used for making a* simple if *instruction*

```
        <genIf>   ::=   'generalIf(' <type>  <evaluation>  <alternative> * <elsePart>  ')'
  <alternative>   ::=    <selection> + '|then' <imperatives>
    <selection>   ::=   '|case' <evaluation>
     <elsePart>   ::=   '|else' <imperatives>
   <evaluation>   ::=    <instruction> *
  <imperatives>   ::=    <instruction> *
  <instruction>   ::=    <singleLineInstruction>  | <MultiLineInstruction>
         <type>   ::=   'boolean' |'char' |'integer' |'real'
                        |'string' |'object' |'pattern'
```

An code fragment using general if is shown in figure 5.18.

```
generalIf(integer
   PUSH-integer {"N/0"}
   RESETFRAME
|case
   PUSHI-integer 1
   RESETFRAME
|then
   ...
|case
   PUSHI-integer 2
   RESETFRAME
|case
   PUSHI-integer 3
   RESETFRAME
|then
   ...
|else
   ...
   RESETFRAME
)
```

Figure 5.18.: *Gbeta byte code fragment using* general if

There is no limit to how many alternatives that can be inserted in a general if instruction so there is no predetermined number of instruction lists in a general if instruction. There will, however, be an instruction list for each <genIf>, <selection>, <alternative>, and <elsePart>, some of which may be empty.

The execution of <evaluation> in <genIf> returns a value of type indicated by <type>. The selections are then evaluated from the top comparing their results to this value. If a match is found the instructions in the associated <imperatives> are executed next and the rest of the <selections> will not be considered. If no match is found the instructions in <elsePart>'s <imperatives> are executed; if no <elsePart> is present, the effect is as if an empty <elsePart> had been present.

Our translation of the example in figure 5.18 is shown in figure 5.19.

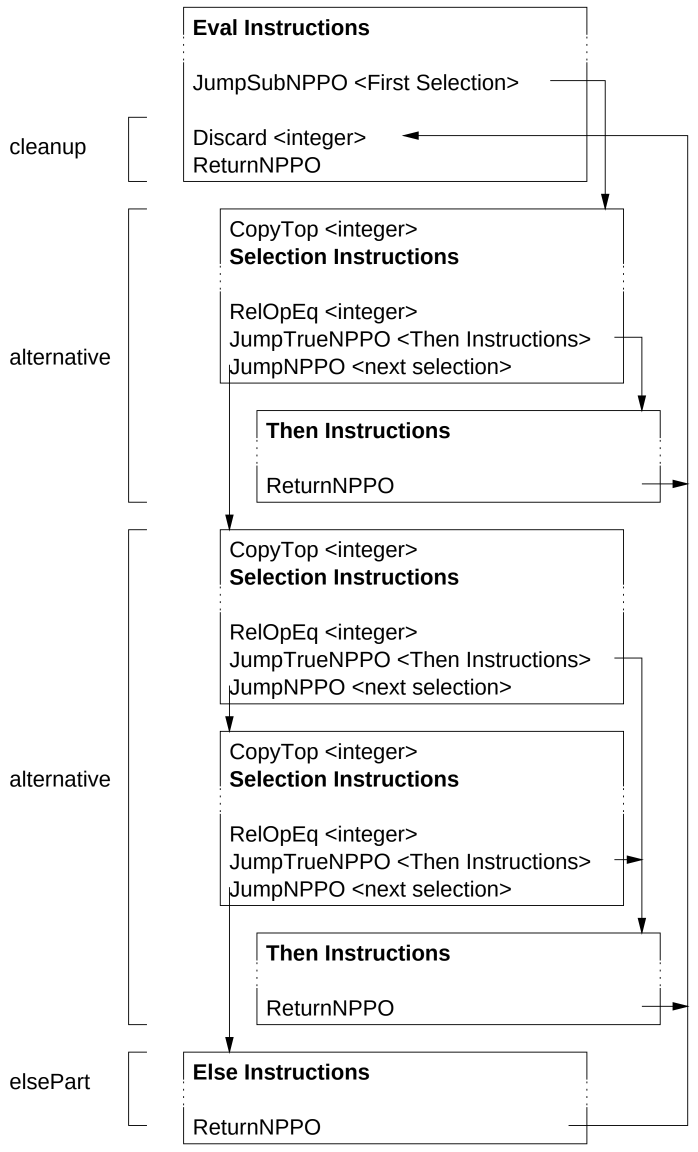


Figure 5.19.: *The instructions used for translating the general if instruction in figure 5.18*

The general if instruction itself only pushes the next instruction onto the execution stack and changes the program counter to point to the first instruction in the evaluation instruction list.

5.5.5. While

The while instruction contains two lists of instructions: an evaluation list and a body list. The evaluation list is executed first. This leaves a value on the boolean stack. If this value is true, the body instructions will be executed and the while loop is restarted from the evaluation. If the value is false, the while instruction is terminated. Our instructions added to the instruction lists are shown in figure 5.20.

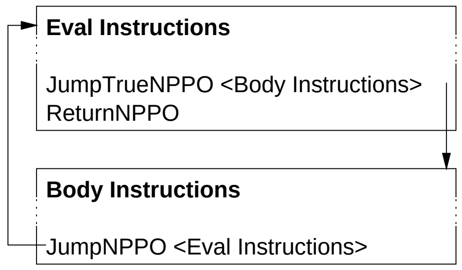

Eval Instructions
JumpTrueNPPO <Body Instructions>
ReturnNPPO
Body Instructions
JumpNPPO <Eval Instructions>

Figure 5.20.: *The instructions used for making a while instruction (for obvious reasons, this was our first multi-line instruction)*

The while instruction itself only pushes the next instruction onto the execution stack and changes the program counter to point to the first instruction in the evaluation instruction list.

5.6. Complex Single-Line Instructions

During the implementation of instructions we were constantly reminded of the great variety in the complexity of different single-line gbeta byte code instructions. In this section we will present some implementation details of the most complex and interesting instructions that were implemented. A description of all gbeta byte code instructions can be seen [Ern99, appendix D].

5.6.1. Add Main-Part

The purpose of the `ADD-mainpart` instruction is to construct new patterns using old patterns, mixins, part objects, and main-parts. The instruction pops a pattern from the pattern stack, creates a new pattern with a mixin list with room for one more mixin than the popped pattern. Then it copies the mixins of the popped pattern to the new pattern and adds one mixin to the end of the mixin list of the new pattern. The `origin` attribute of this mixin must point to the part object specified by the run-time path and its `mainpart` attribute must point to a `MainPart` having the main-part id given as argument to the `ADD-mainpart` instruction. To support finding this main-part we have implemented a `getMainPart(mainPartId)` method on the `Scheduler` class. This method just searches its list of main-parts and returns the main-part having main-part id `mainPartId` if found. This rather expensive iteration could have been avoided with a two pass parser which could add pointers to the `ADD-mainpart` instructions pointing to the correct main-part. If we had placed the main-parts in a hash-map data structure we could have achieved constant average lookup time instead of linear (in the number of main-parts) lookup time.

5.6.2. Merge

The `MERGE` instructions use an implementation of the `merge()` function described in subsection 2.1.2 to pop two patterns from the pattern stack, merge them if possible, and push the resulting pattern to the pattern stack. The pattern merging is one of main concepts separating gbeta from BETA. To implement the merge operation soundly two special handling cases were introduced. In figure 5.21 the implemented `merge()` function is presented. The help function `member(mixinList1, iterator, mixinList2, mixinList2Length)` returns true if the element at `iterator` in `mixinList1` is a member of `mixinList2`.

SETUP Gets the mixinlists and length of the patterns and creates iterators for the patterns and the result pattern

SPECC. Handles the special case that one or both of the patterns is the empty pattern. If `p1` is the empty pattern, `p2` is returned and vice versa. If both patterns are the empty pattern, `p2` is returned.

THE MERGE ALGORITHM Resembles the merge algorithm described in subsection 2.1.2.

RESULT When a merge of the patterns succeeds a new resulting pattern is created and pushed to the pattern stack. To get a correct `ObjectDescriptor` we have a global pattern objects-descriptor array which is lazy initialized, i.e., an `ObjectDescriptor` supporting a pattern with the length e.g., 7 is created the first time it is needed.

5.7. Virtual Machine Evaluation

Since the focus of this project is memory management a number of possible optimizations and extensions to the virtual machine component have been omitted. In this section we evaluate the current implementation and propose a number of improvements.

5.7.1. Static and Dynamic Strings

In this implementation strings is placed in the non-traced root space, so if a string becomes dead we waste the space it uses. Instead of putting all strings into non-traced root space we could have had two types of `PredefinedStringPartObjects`: dynamic strings and static strings. The dynamic strings could have been allocated in train space and garbage collected if dead. Dynamic strings would be allocated because of the `Stdio/in` instruction. We would still have to allocate the strings parsed from the gbeta byte code in the non-traced space, since they should be kept alive until program termination (it is impossible to know when an instruction will not be executed again according to the halting theorem and its siblings [Sip97]).

This extension to the virtual machine would be very easy since the only thing it requires is two types of `PredefinedStringPartObject` object descriptors: one for the dynamic strings indicating a pointer to the `VMObject` containing the string, and one indicating no pointer for the static strings.

5.7.2. Multi-threaded

To make our virtual machine multi-threaded a number a things should be changed to both the virtual machine component and the memory management component. Since concurrent threads allocate different gbeta entities concurrently, it is vital that some sort of resource sharing is present in the memory management system. Also, the `Scheduler` should have some sort of signaling queue to allow for different signals to be queued and eventually processed. This would amongst other things allow safe points with concurrent threads. I would be possible to use native threads with few modifications of the virtual machine component, but the memory manager requires larger modifications.

```
merge(Pattern_t *p1, Pattern_t *p2){
S  Mixin_t **p1MixinList = p1->getMixins(), **p2MixinList = p2->getMixins();
E  int p1Length = p1->getLength(), p2Length = p2->getLength();
T  int p1I = p1Length-1; //iterator for p1
U  int p2I = p2Length-1; //iterator for p2
P  int resI = 0;         //result iterator
   //check for EmptyPattern
S  if(!p1MixinList[0]){
P    return p2;
E  }
C  if(!p2MixinList[0]){
C     return p1;
.  }
   //merge p1 & p2
T   bool notP1Finished = 1, notP2Finished = 1;
H   while(notP1Finished || notP2Finished){
E    if((p1MixinList[p1I]->origin == p2MixinList[p2I]->origin) &&
        (p1MixinList[p1I]->getMainPart() ==
M         p2MixinList[p2I]->getMainPart())){
E      mergeScratch[resI++] = p1MixinList[p1I];
R      if(p1I){
G        p1I--;
E      } else {
         notP1Finished = 0;
A      }
L      if(p2I){
G        p2I--;
O      } else {
R        notP2Finished = 0;
I      }
T    } else if(!(member(p2MixinList, p2I, p1MixinList, p1Length)) &&
H              notP2Finished){
M      mergeScratch[resI++] = p2MixinList[p2I];
.      if(p2I){
.        p2I--;
.      } else {
.        notP2Finished = 0;
.      }
.    } else if(!(member(p1MixinList, p1I, p2MixinList, p2Length)) &&
.              notP1Finished){
.      mergeScratch[resI++] = p1MixinList[p1I];
.      if(p1I){
.        p1I--;
.      } else {
.        notP1Finished = 0;
.      }
.    } else {
.      return NULL;
.      break;
.    }
.  }
   //create new pattern for result
R  ObjectDescriptor_t *ptnDesc = getPtnObjectDescriptor(resI);
E  Pattern_t *result = new(allocateVM(ptnDesc)) Pattern_t(ptnDesc);
S  Mixin_t **resultMixinList = result->getMixins();
U  //copy result from scratch to new pattern
L  for(int i = resI, j = 0; i > 0; i--,j++){
T     setVMReference(result, (void**)&resultMixinList[j], mergeScratch[i-1]);
.  }
. return result;
}
```

Figure 5.21.: *The implemented* `merge()` *function.* **SPECC.** *abbreviates special case handling*

5.7.3. Two-Pass Parsing of the Byte Code

If we had made the `ByteCodeLoader` a two-pass parser a number of optimizations could have been obtained:

Cached Main-Parts `AddMainPart` instruction could have main-part pointers cached yielding constant lookup time.

Real Byte Code Format The instruction lists in main-parts could be parsed to a real byte code format, i.e. with one-byte opcodes and zero or more argument bytes. This would make the byte-code lists smaller, but the real benefit would be the possibility to interpret the byte codes faster.

5.7.4. Statically Known Patterns and Objects

Recent versions of the gbeta compiler includes information about statically known patterns and objects in the gbeta byte code files. The structure of a *static pattern* and a *static object* is known at compile time, but these entities cannot be constructed before run-time. In cases where a method is not invoked a great number of times, it can save space and initialization time to use static patterns, as they do in [JJW01], instead of only creating them dynamically.

6. The Memory Manager

In [IP01] we made an almost fully working memory manager which used the train algorithm to garbage collect but did not have other generations. We proposed a number of changes to our first design and implementation. In essence the main goal of our proposals was to move the focus from 'easy to understand' to efficient [IP01]. We will now present the subjects considered for the new version of the memory manager.

Efficient Write Barrier The implemented write barrier had a worst case time complexity of $O(n+m)$, where n was the maximum number of objects in a car and m was the number of objects referencing another object. Also, the space complexity was $O(n)$ where n was the number of pointers in `VMObjects`.

Using a more efficient remembered set data structure (e.g. a hash set) and aligning cars at 2^k-byte boundaries will induce a write barrier typically at constant time except in rare cases where the hash map needs resizing or collisions occur.

No Car-Internal Remember-References Car-internal remember-references made our garbage-collection algorithm easy to understand since the precise knowledge of all interesting pointers made Cheney scans unnecessary. But these car-internal remember-references both increased the space used by the remembered sets and made the write barrier more expensive than necessary.

Instead of having car-internal remember-references, we could evacuate followers from the from car using scan pointers. These scan pointers should point into the cars where externally referenced objects have been moved.

Reduce `VMObject` Space Overhead Aligning cars at 2^k-byte boundaries also makes it possible to remove the car pointer in `VMObjects`, since the address of the car can be obtained by clearing the k-least significant bits of `VMObject`'s address. The only two attributes needed in `VMObjects` is an object descriptor pointer and a forward pointer.

Introduction of New Objects In the old memory manager, new objects were born in the car with the highest train and car number except the root objects which were born in a special train. Since young objects are most likely to die, this is not the optimal strategy for a garbage collector.

A special object space, called the introductory space, for introducing new objects could reduce the collection overhead because less garbage would be dragged through the whole system. Some kind of write barrier is required to track pointers from the train-managed heap to the introductory space. This means that whenever a car has to be scavenged, the introductory area has to be scavenged too because it potentially holds references from live objects into the first car. This scheme is in fact be a small generational garbage collector.

Popular Objects As it is expensive to move objects which are referenced a lot, some way of handling popular objects would be a good optimization. One way of doing this is to avoid moving the popular object by keeping the car after the scavenge and reassign the train reference and car number [GS93].

Large Object Handling The maximum size of objects is bounded by a fixed fraction of the size of a car. Since cars also have a fixed size, objects cannot grow arbitrarily large. A large object space could solve this problem [GS93]. Using a large object space can also reduce the garbage collection overhead because moving large objects is expensive.

Concurrency Multi cpu architectures support would be a great benefit for the virtual machine. If this has to be added to the virtual machine, a couple of time critical functions, such as the write barrier and allocation function, have to be able to handle multiple threads. This requires either synchronization of these routines or a very clever design to avoid that. Concurrent garbage collection is also desirable but probably even harder to implement efficiently.

Adaptiveness It would be interesting to make an adaptive garbage collector which adapted its garbage collection frequency to the amount garbage on the heap. This is a feedback system problem [FPEN94]. In feedback systems it is vital that you control the process. A good example is a pot of almost boiling water. If one continuously controls the amount of energy transferred to the pot one can control the temperature of the water and thereby also avoid that the water boils. The same goes for controlling the garbage collection frequency. If one garbage collect too much one wastes time, but if one garbage collects too little one wastes space. Using feedback theory [FPEN94] we end up with a situation as depicted in figure 6.1. But this is not the whole truth! Since the initiations of our garbage collector is event-based, we need an discrete event-based feed-back control system [PB98] to control it. To simulate discrete event-based feed-back control systems a variant of Petri net models must be used and a so-called Lyaponov framework can be used to model these. This is far off the subject of this report but the idea might be worth pursuing, and it could be subject to future work.

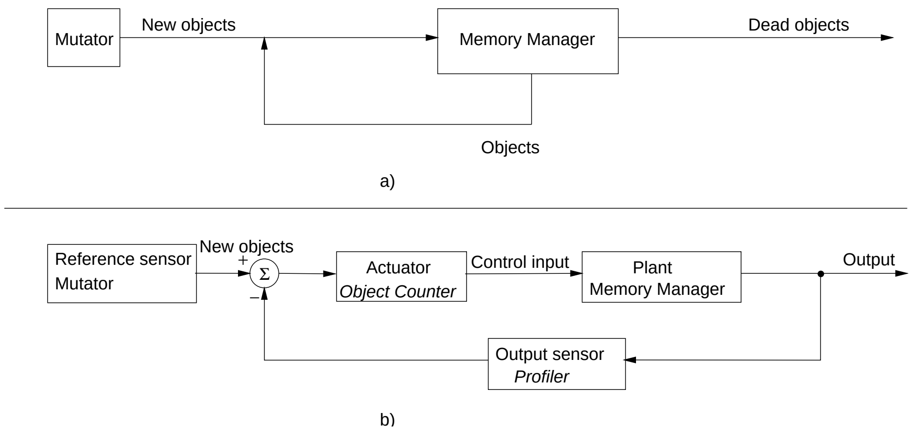


Figure 6.1.: *a) Our system without profiler, b) Our system as a simple negative feedback loop block diagram - Texts in italic shows new components needed to make it adaptive*

We have decided to implement a new memory manager with the above improvement proposals except the last four (popular object handling, large object handling, concurrency, and adaptiveness). It will still be an accurate, handleless, partial, cooperative, and single threaded garbage collector.

The purpose of this new implementation is to make a reasonably efficient train algorithm garbage collector that allows us to experiment with different aspects of the train algorithm such as strategies to the introduction of new objects and the performance effect of varying the car and the introductory space sizes.

Besides the above proposals we have decided to make a special stack space to circumvent the write barrier when writing to reference stacks. This was inspired by [Mos87].

This chapter will document the design and implementation of the new version of the memory manager. The first section will give an overview of the architecture. The following three sections will give a more in-depth explanation of the three spaces present in the heap. The next three sections describe the remembered set implementation, the object descriptors, and the redesign of the object headers used for objects in the virtual machine. Then a section describes how the garbage collection algorithm has been implemented in the system. Next, the write barrier is explained, and finally the interface is described.

6.1. Architecture

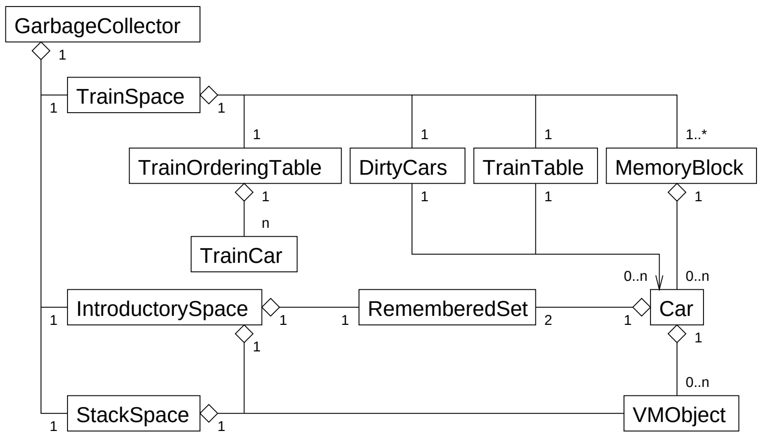


Figure 6.2.: *Class diagram of the memory manager*

Figure 6.2 shows the class diagram of the memory manager. For those interested in the source code included on the disc, we will mention a few diversions from this class diagram. The `TrainOrderingTable` is not a class in its own right but included in the `TrainSpace` class which by the way is called `Train-Generation` in the source. Also, the class `TrainCar` was unfortunately called `CarTrain`. The code is placed in the directories `gbvm/src/gc_new` and `gbvm/src/gc_common`. Most of these classes will be described more in-depth in the later sections. We will now move on to present the heap layout used.

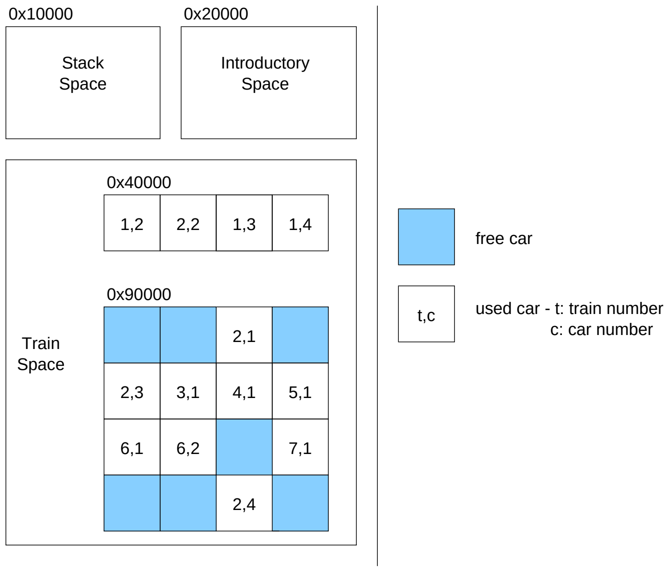


Figure 6.3.: *The heap layout. The two blocks of cars in the train space represents* `Memory-Blocks`.

The heap is divided into three spaces: the stack space, the introductory space, and the train space as illustrated in figure 6.3.

The *stack space* is where all the stacks carrying references are allocated and other objects which will have to be scanned during garbage collection for determining the root set. The space is not scavenged because only one thread is supported. If the stacks of the thread are garbage the thread is garbage. and the program execution has terminated. Stacks cannot be resized requiring allocation of new memory and freeing of old memory.

The *introductory space* is the place where the majority of new objects start their life. If they are still live when the next scavenge occurs, they will be moved to the Train Space.

The *train space* is where dynamically allocated objects live after they have survived the first garbage collection.

In addition to these spaces untraced root objects are allocated using the standard `malloc()` function.

6.2. Stack Space

The main motivation for having a stack space, compared to having stacks floating around in the train space, is that when writing to stacks containing references, the write barrier can be circumvented, which yields lower mutator overhead related to memory management. Instead of using the write barrier for remembering references from the stack space, the stack space is scanned at each garbage collection. This identifies live objects in the introductory space or the from car (if both the introductory space and from car is being garbage collected).

Another advantage of the stack space is that the large stack objects do not need to be moved making garbage collection faster.

The stack space is an optimization that is particularly worthwhile with mutators that use many reference stack operations such as our gbeta virtual machine (see chapter 5).

The reason for storing the stacks in the same place is that we want to be able to scan them for references easily. This also means that the stacks, which do not contain any references, should not be placed in this space since that would only incur extra scanning overhead. Such reference free stacks are allocated using the standard `malloc()` function outside any of the three spaces. Actually, the objects left in the stack space constitute the root set of the live object graph.

6.3. Introductory Space

The purpose of the introductory space is to host new objects. There are two advantages gained when allocating new objects in a separate space.

Firstly, since many objects die young, it is beneficial for the efficiency of the garbage collection to be able to concentrate its effort on the introductory space.

Secondly, if the introductory space is garbage collected with the train space, the write barrier overhead of stores in introductory space objects can be reduced to only identifying the case of a store in the introductory space; no remembered set update is necessary. This is consistent with common generational garbage collectors that only remember references from older to younger generations. To ensure this in the implementation the introductory space is always logically ordered lower than anything in the train space, effectively making the introductory space a special train with only one car that is always collected with the first “real” train.

To keep track of interesting pointers into the introductory space, a remembered set contains the slot of a pointer from train space into introductory space. This remembered set is identical to the remembered sets of a car. Remembered sets will be discussed in section 6.5.

In order to have a fast membership test for this space, the memory is allocated on a 2^k aligned address where 2^k is the size of the memory allocated for the introductory space. This way it is possible, with a bitwise AND operation to remove the lower *k* bits of a given address and compare this to the base address of the introductory space memory.

6.4. Train Space

The purpose of train space is to host mature objects and support an efficient and non-disruptive garbage collection of these objects using the train algorithm.

6.4.1. Memory Blocks

Memory blocks manage the raw memory allocated from the operating system. A memory block contains a number of car sized blocks that can be reserved for cars. Each of these sub blocks (or cars) are aligned on an address divisible by the size of a car. The space within a memory block is managed by a free pointer and a free list. The free list contains the recycled cars, while the free pointer is used for allocating space for a new car within a memory block (see figure 6.4).

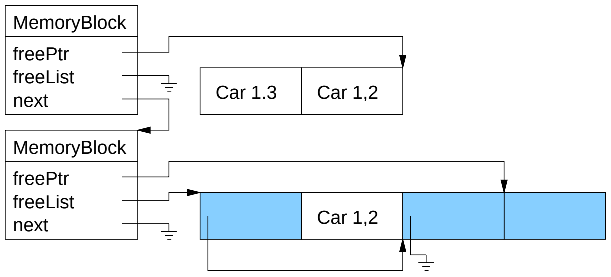


Figure 6.4.: *The memory blocks managing the allocation of car space*

When one memory block is filled a new one is allocated, but this time a constant factor *k* larger than the previously largest block. This way the sizes of memory blocks grow exponentially. The factor *k* is typically in the range 2 to 4. When several memory blocks are active, allocation of a car will first be attempted in the smallest memory block and last in the largest. Although the likelihood of finding an available car this way is smaller than if the largest was searched first, it will be possible to free the largest block at some time, should the memory requirements decrease later.

6.4.2. Cars

The class `Car` is shown in figure 6.5. Cars host mature objects. Each car has a `nextCar` pointer pointing to the car next in the train, a `freePtr` pointer pointing to the first free slot in the car, and a `scanPtr` used for Cheney scanning. Besides that we have a pointer to a train-internal remembered set, and a train-external remembered set. Finally the cars have a data array which in the implementation (in `C++`) is declared to have the size of one void pointer which, like the `VMObjects`, is actually a lot larger and is the place where the `VMObjects` are stored.

The car has both a train internal and a train external remembered set, since it is vital for the efficiency of the train algorithm to be able to quickly determine whether there are any train-external remember references or not. That is, to ensure that a train can be deleted it must be efficient to check if there are any pointers into the train from outside the train. If the remembered sets were merged, it would be much more expensive to handle large garbage structures filling several cars in the from train since every train internal remember

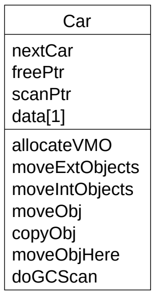


Figure 6.5.: *The `Car` class*

reference would have to be checked before it was possible to conclude that the train only contained dead objects.

The `Car` class offers methods to allocate new objects, move objects, methods to access the remembered sets, and a method to do a Cheney scan of the car. Since `Car` is one of the very important classes in the memory manager, we will give a brief overview of its most important methods in the following paragraphs.

The method `allocateVMO()` makes it possible to allocate new objects in a car. This is only done rarely, namely when the introductory space is filled up.

There are several methods for moving objects in the `Car` class. The two first, `moveExtObjects()` and `moveIntObjects()`, are only used when a car has the role of from car. The methods traverse their respective remembered set moving objects being referenced to the place where they are referenced from. For this task the method `moveObj()` is used. The main task of `moveObj()` is to determine the exact place to copy an object. Using the `copyObj()` method, first the referencing car is tried, then the last car in that train, and finally a new car in that train. If all this fails, the object is too large to fit inside a car and the program execution is halted.

`copyObj()` is the main interface for moving objects to a car. It takes care of things such as checking for a forward pointer, checking if there is enough space for the object, setting the forward pointer, updating the car's free pointer, and dirtying the car (see subsection 6.4.4).

The function `moveObjHere()` is another interface to moving objects to a car. It is used when an object should only be put in a specific car and only if that car is not filled above a fixed *fill-threshold*. If that is not possible, the method fails. This method is only used when moving objects that are only referenced from the stacks, and only with our new train creation policy (see subsection 6.8.2 and section 7.6). The implementation is very much like the `copyObj()` method.

The final method we will mention is `doGCScan()` which is the method performing the Cheney scan on a car. This method traverses all references in the objects between `scanPtr` and `freePtr`. If the reference points to an object inside the introductory space or the from car, `moveObj()` is invoked for moving the object to this car or retrieving the forward pointer, and `setVMReference()` is used for setting the reference and updating the remembered set if necessary. If the reference refers to an object outside introductory space and from car, the object is not moved, but the remembered set is still updated if necessary.

6.4.3. Trains

Instead of having trains as a class, like we did in our earlier versions of the garbage collector, we manage trains using a car-ordering table, a train-table, and by linking the cars in a singly-linked list.

Train Table

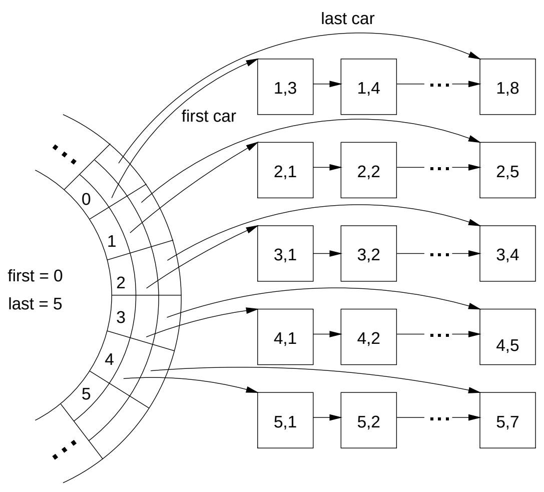


Figure 6.6.: *The train-table. First denotes the first train (the from train) and last denotes the first free cell in the circular buffer*

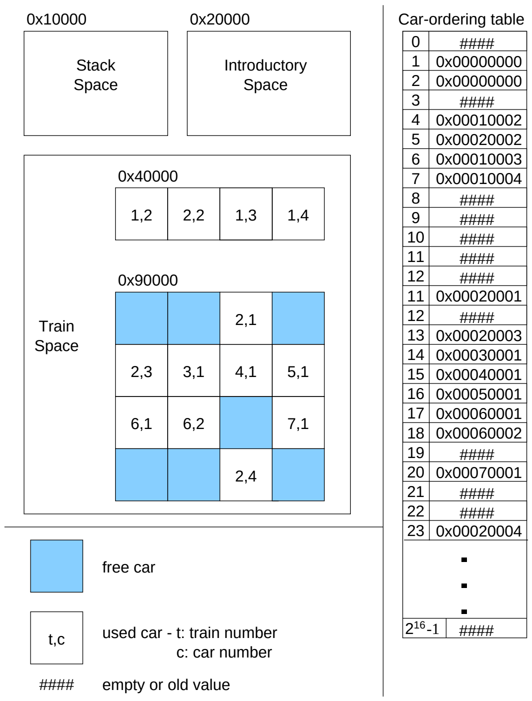
The train-table is used during garbage collecting to get the first or last car of a train given a train number. It has been implemented as a circular buffer, which has constant lookup time and little book-keeping. However, this data structure limits the number of concurrent trains. To circumvent this, dynamic resizing of the buffer would be necessary or a circular buffer with the size of the number of possible trains could be allocated. The implemented train-table is illustrated in figure 6.6. To return the last car given a train number *t*, the `TrainTable` class uses *(t - 1)* modulus *bufferSize* as index into the circular buffer. This means that when train numbers exceed the highest possible number with the given buffer size, they will begin at the start again. This works in the train algorithm because the lowest order train is continuously removed making space for new trains, but only as long as the number of trains does not exceed the size of the buffer.

Car Ordering Table

The task of the car-ordering table is to keep the total ordering of cars in the train algorithm. In addition everything outside the train space (in particular the introductory space) is put into this ordering as the lowest order element. This is beneficial as it simplifies the tests needed when updating remembered sets in the write barrier, as we will show in section 6.9.

When given an address, the car-ordering table returns an object containing a 32-bit integer where the 16 most significant bits are the train number and the 16 least significant bits are the car number - we refer to these objects as *train-car elements*. This way we can compare the ordering of two cars efficiently. The car-ordering table is illustrated in figure 6.7. To return a train-car element given an address, the memory manager simply shifts the address 16 bits right and uses this value as index in the car-ordering table. This resembles the card marking scheme [WM89]. By initializing all values of this table to 0, everything outside the train space is automatically placed in train 0, car 0 so the lowest legal train number in the train space is therefore 1.

The size of the car-ordering table is fixed to a size so it is able to cover the whole address space. With a car size of 64KB this results in a 256KB table. The train-car elements do not adapt to this increased number of cars, only 65536 cars (in each train) and 65535 trains are possible. This does not seem to be a restriction in any of the tests we have run but with smaller car sizes and more memory available it could be. With a



0	####
1	0x00000000
2	0x00000000
3	####
4	0x00010002
5	0x00020002
6	0x00010003
7	0x00010004
8	####
9	####
10	####
11	####
12	####
11	0x00020001
12	####
13	0x00020003
14	0x00030001
15	0x00040001
16	0x00050001
17	0x00060001
18	0x00060002
19	####
20	0x00070001
21	####
22	####
23	0x00020004
. . .	
2^{16}-1	####

Figure 6.7.: *The car-ordering table. #### means that the value of this entry is either empty or old and not interesting*

very low car size of 1KB the approximate theoretical maximum of bytes in a train would be 65MB, and with the lowest size we have tried, 4KB, it is 256MB. Using some of the bits currently used for the train number could solve this for some time.

As the train number continually grows, it is sometimes necessary to renumber the trains which means that a new lowest train number must be established and every train-car element of the cars in the current trains must be updated. This can probably be quite expensive with large heaps, but it happens very rarely in practice. To avoid a reordering of the trains in the train-table, the new train number of the lowest train is chosen as the lowest that will result in the same index in the train-table.

6.4.4. Dirty Cars

When garbage collecting, it is necessary to know to which cars unscanned objects have been moved. Unscanned objects resemble the objects between the scan and the next pointer in a copy collector using a Cheney scan. These are commonly referred to as grey objects in tricolor marking schemes [Wil94]. In the train algorithm unscanned objects are the objects moved from the introductory space or the from car because of external references, i.e., references from other cars, the stacks, or other root objects. We refer to *dirty cars* as cars with unscanned objects.

A hash set named dirty cars always contains references to the current set of dirty cars. Using a hash set, duplicate references are avoided and insertion is a cheap operation. This is necessary during garbage collection where many objects are moved. As long as this set is not empty, the garbage collection is not finished.

6.5. Remembered Sets

In [IP01, p81-83] we found that remembered sets were more suitable than card marking for the write barrier within the train algorithm. Some other virtual machines using the train algorithm [SM01, Hud00] combine card marking with remembered sets (see chapter 8), but since the write barrier implementation is not the main focus in this project, we have chosen only to use remembered sets.

A write barrier in the form of remembered sets is used to keep track of interesting pointers, i.e., pointers from a higher ordered car to a lower ordered car, and pointers from train space into introductory space. We have chosen to remember slots instead of objects [HMS92] because this frees us from the task of searching the objects for the pointer, and there is not much use for the object header either. That is, we remember the address where an interesting pointer is situated on the heap directly. This is illustrated in figure 6.8

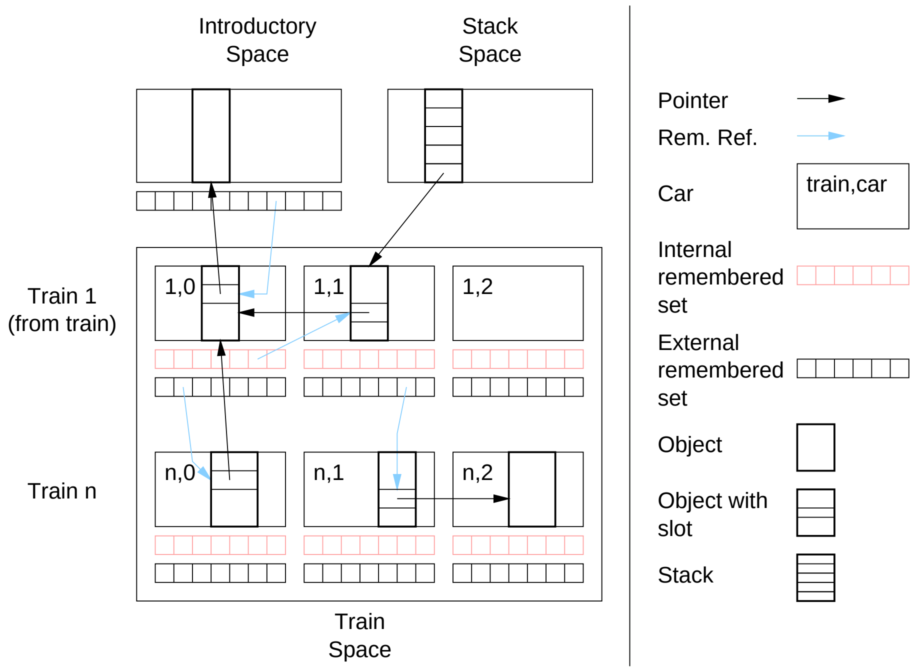


Figure 6.8.: *Remembered sets in the heap*

It is interesting to note the remember reference from car (1, 1) to the slot in car (n, 1). The pointer in this slot does not point back at an object in car (1, 1), but this is just a fact of life with this implementation of remembered sets, one cannot be sure that the remembered slot still refers to the object it did when it was created, this must be checked when using the remembered set entries. Removal of these dangling entries would cause extra overhead in the write barrier, and the remembered set implementation would have to be able to handle deletions.

The remembered sets were first implemented using a hash set from the Standard Template Library, STL, available with most standard `C++` compilers, and it is still possible to use this implementation. In order to be sure about what actually happens when the remembered sets are used, we made our own hash set implementation. The implementation is not that interesting so we will only present a few details. The primes for the array sizes are identical to the ones STL implementation uses. Growing (to about the double size) happens when the fill fraction reaches a fixed threshold (currently 0.75). Shrinking never happens because deletions from the set is not possible. When inserting and a collision happens, the array is linearly searched until a free entry is found, possibly wrapping around if the end is reached.

6.6. Object Descriptors

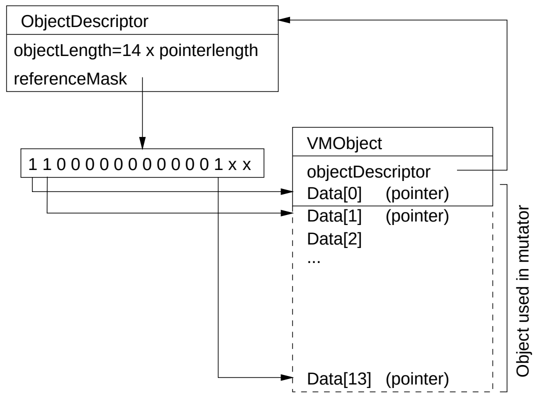


Figure 6.9.: *The relation between object descriptors and an object. For simplicity, the figure shows a reference mask with only one 16-bit integer (the x means that their value is unimportant since these are excess reference indicators)*

The purpose of object descriptors is to hold information about the objects used by the mutator which is needed when garbage collecting. Object descriptors are unrelated to the syntactic category <Object-Descriptor> in the gbeta grammar [Ern99]. The object descriptor holds length and reference placement information about `VMObject` instances (see figure 6.9) which is enough information for an accurate garbage collector.

Objects are indexed as an array of pointers, so with current 32 bit architectures, pointers would typically have to be four byte aligned. Data fields smaller than the size of a pointer are still possible in the mutator as long as four byte pointer alignment is preserved. The object descriptors represent each 32 bit field in an object with a bit in a reference mask indicating whether the field holds a reference or not.

The reference mask structure for representing references is space efficient when there is a high density of references in objects but less space efficient if objects are generally large with few references. Since our object descriptors use space proportional to the length of the object, it might be appropriate to choose another design in that case. This could e.g., be a zero terminated array of `VMObject` base offsets indicating where the references are in the object, since this design would use space proportional to the number of references instead. However, this is probably not a real issue now as pointers are very common, objects are small in average, and object descriptors are highly shared structures.

Alignment of our data is another issue the `ObjectDescriptor` is involved with. Since an incorrectly aligned pointer can cause a bus error, the `ObjectDescriptor` makes sure that whatever size is requested, the actual size stored in the `ObjectDescriptor` is always divisible by four, rounding up if necessary. This way the allocation system can always add sizes originating from `ObjectDescriptors` without the concern of creating an incorrectly aligned pointer. This ensures that `VMObjects` will always be four byte aligned as well as the 32 bit fields inside the `VMObjects`.

6.7. VMObject Layout

The object layout has been changed a lot in the new memory manager in order to reduce the space overhead. This has only affected the header information stored in the `VMObject` class used by the garbage collector; the object used by the mutator is still placed in the data array, making the change practically invisible to the mutator. In the old implementation, where no effort had been done to reduce the header, the header used

28 bytes on an 32-bit machine, whereas the new implementation only needs 4 bytes or 8 bytes if the object has been moved (see figure 6.10). The reason, we have done such an effort to reduce this space overhead, is that early experiments with space waste showed that about one third of the space used for `VMObjects` was consumed by headers with an implementation where a header filled 12 bytes. Below we will explain the transition from the old to the new object layout, field by field.

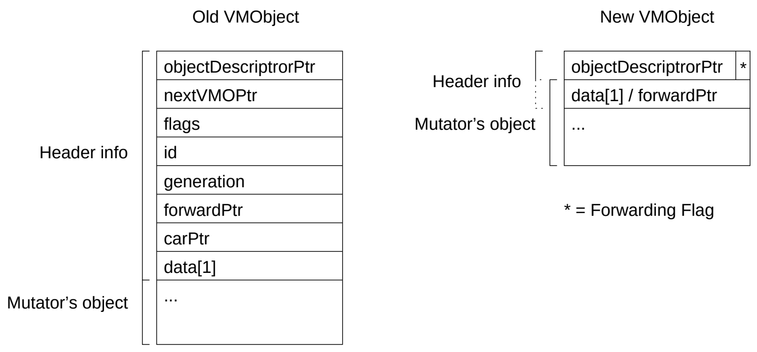


Figure 6.10.: *The old and the new vmobject*

`objectDescriptorPtr` is still necessary because the garbage collector still needs to know information about the object such as its length and where it has references. This information is of variable length and can be shared among similar objects. For this reason it is practical to separate it from the actual object.

The least significant bit of the pointer is used for a forwarding flag, so this bit is always cleared when the `objectDescriptorPtr` is requested. A more thorough description of the forwarding flag is found in the `forwardPtr` attribute discussion.

`nextVMOPtr` is unnecessary because it can be calculated from the length information available in the object descriptor and the fixed size of the header. The last object in a `Car` can be identified by comparing the calculated `nextVMOPtr` against the free-pointer of its `Car`.

`flags` was a leftover from the distant past. It was forgotten for a long time and unused in the old prototype.

`id` is only used during debugging. It was mainly useful when the garbage collector was tested alone as it enabled us to identify objects by a number. In the current version it can be enabled, but it is disabled by default.

`generation` was never used since we did not implement a generational garbage collector.

`forwardPtr` is only used during garbage collection after an object has been moved. When an object has been moved, the contents in the old location will never be accessed again so we choose to use the first field of the object for a forward pointer. We still need to know when to interpret this field as object data and when to interpret it as a forward pointer, though. Since the pointer to the object descriptor always has its two least significant bits set to zero because of alignment, we used the least significant bit as a flag indicating whether the first data field of the object is data or a forward pointer.

`carPtr` Because `Car` instances are now aligned on 2^k addresses where k is an integer, it is possible to convert a pointer to anything inside a `Car`, into a pointer to a `Car` simply by setting the k least significant bits to zero. The old implementation did not have aligned `Cars` so this was not as easy at that time.

`data array` This is where the actual object data is stored. Although the size is set to 1 in the class declaration, our allocation scheme makes sure a number of bytes indicated by the `ObjectDescriptor` following the `VMObject` are reserved for an object starting in `data[0]` of that length.

As mentioned in the `forwardPtr` attribute discussion, the first field of the data array is also used for a forwarding pointer when scavenging in the new implementation.

6.8. The Garbage Collection Algorithm

In this section we will show how we have incorporated the train algorithm in our garbage collector. First the pseudo code is shown and then a detailed explanation follows.

```
evacuate train-externally referenced objects from from-car
Cheney scan all dirty cars
evacuate externally referenced objects from the introductory space
evacuate all objects in introductory space or from car
  referenced from the stack space,
  checking if other objects in cars in the from train are referenced
Cheney scan all dirty cars
if( other cars in from train were not referenced from stacks
   AND all other cars in from train have empty remembered sets) {
    reclaim from train
} else {
    evacuate train-internally referenced objects from from-car
      to the car holding the referencing object or last car
    Cheney scan all dirty cars
    reclaim from-car
}
reclaim used introductory space
```

Figure 6.11.: *The train algorithm in our context*

First of all, it is important to note the order objects are evacuated from the different spaces. The stack space references do not give any clue to where an object should be moved. In contrast to this the remembered sets contain information about what object slot and thus what car the object is referenced from. This means that whenever an object can be moved using a remembered set, this should be preferred over moving an object because of a stack reference; it is best to use the remembered sets before the stacks and other root objects.

6.8.1. Introductory Space and From Car Scavenging Order

Another much more subtle issue is selecting the order in which the remembered set of the from car or the introductory space should be used. One might think this does not matter. However, there are situations where it does. Consider the scheme shown in figure 6.12.

If the introductory space is scavenged first, the reference from object 2 to object 1 would indicate that object 1 should be moved to train 1, the from train, but that is not permitted according to the train algorithm. Actually, the train algorithm does not specify where it *should* be moved, only where it *should not*. So as designers, we are left with the choice of where to put the object. One idea would be to put the object in the last car of the last train, and another to put it as close as possible to the object we know that has a reference

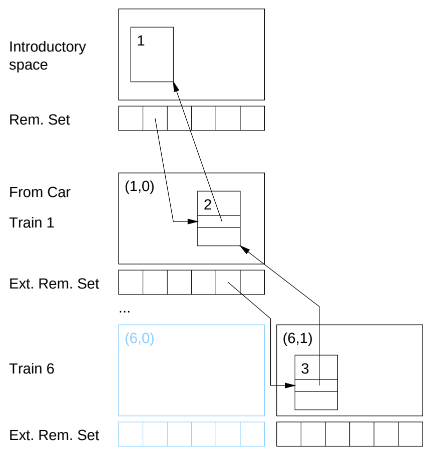


Figure 6.12.: *A snapshot of a hypothetical heap before garbage collection*

to the object. A few quick experiments revealed no obvious advantage of either of these strategies. When the remembered set of the from car is used next, object 2 will be moved into car (6, 1) or at least into train 6. Given a high number of cars and trains, it is unlikely that the objects 1 and 2 end up in the same car or even train, given our strategies for moving objects like object 1.

If, on the other hand, the from car is scavenged first, a Cheney scan is performed, and finally the introductory space is scavenged, all the objects could potentially end up in car (6, 1), and they will certainly end up in train 6. There is still a drawback of having to scan the remembered set of the introductory space after the remembered set of the from car has been scanned; some of the entries in the remembered set may refer to objects that have already been moved, *zombie objects*. This is never the case when the introductory space remembered set is scanned before the from car remembered set because a reference from the introductory space to the train space is not remembered in any remembered set.

When the remembered set of the introductory space is scanned, both objects 1 and 2 have been moved, and one of two things can happen.

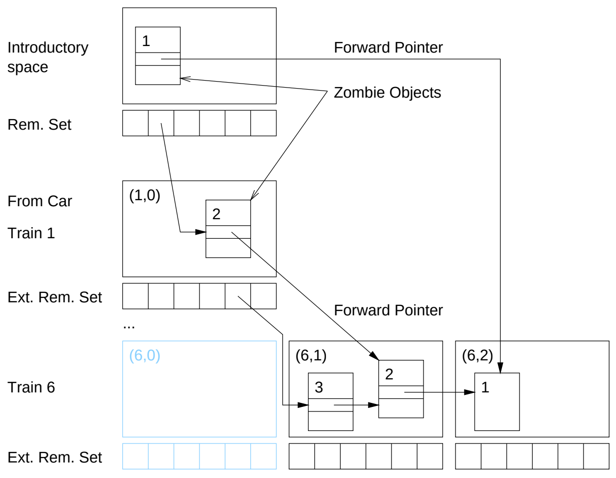


Figure 6.13.: *The hypothetical heap after scavenging car (1,0), Cheney scanning, and scavenging the introductory space*

First, if the reference from zombie object 2 to zombie object 1 has been overwritten by a forward pointer[1], it cannot point into the introductory space anymore since objects are never moved there, and the implementation will just proceed with the next remembered set item (see figure 6.13).

Second, if the pointer in zombie object 2 still points at zombie object 1, the algorithm will try to move zombie object 1 somewhere, but since zombie object 1 has already been moved, it contains a forward pointer, and this will be written to the slot in zombie object 2 (see figure 6.14). Note that it is not trivial to see that zombie object 2 is in fact a zombie object since the remembered set only contains a pointer to the reference slot, not the object base, but writing this forward pointer from zombie object 1 into the slot of zombie object 2 is not an expensive operation. Also note that the real object 2 already has the correct value of this pointer as this was set during the Cheney scan that also moved object 1.

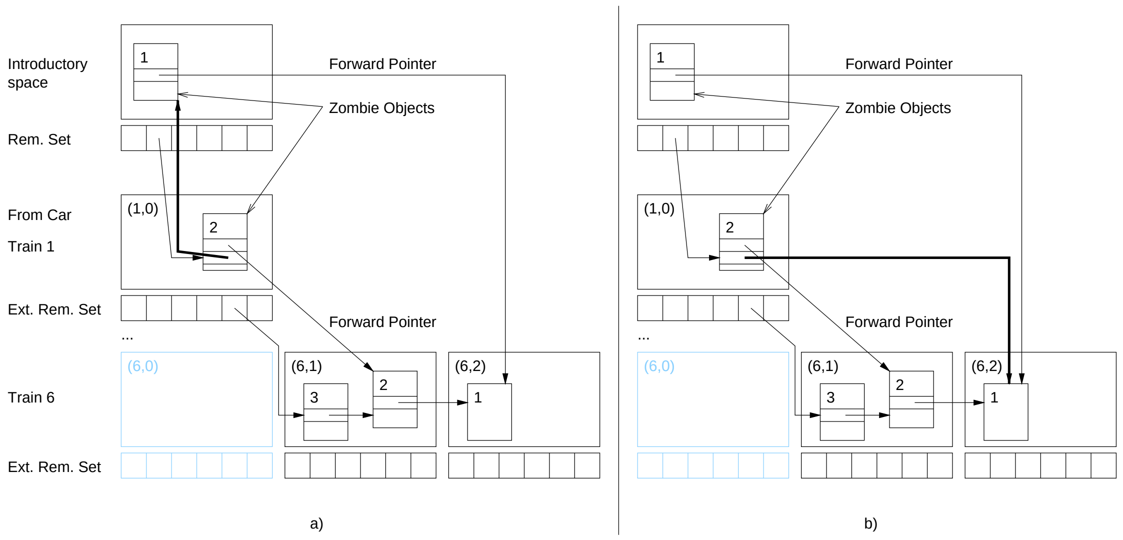


Figure 6.14.: *a) before scavenging the introductory space b) after scavenge of the introductory space; the pointer in zombie object 2 has unnecessarily been updated.*

A modification of the write barrier could also filter out the remember references from the from car to the introductory space causing the above situation. This makes the write barrier slower for all the stores in the train space to the introductory space except for those in the from car where it would be faster. In our opinion none of the above two situations justify such a change as they seem to be harmless.

Some practical experiments show a minor speed increase when the first car is scavenged before the introductory space when compared to the opposite order.

The major part of the problem of finding a suitable location for an introductory space object referenced from the from train still remains, though. The from car is only a special case of any given car from the from train, and it is impossible to do the same thing when an introductory space object is referenced from another car in the from train, at least without scavenging the whole train. A possible solution could be to remember the objects referenced from the from train and postpone their evacuations until every remembered set item has been checked and then reconsider the set of postponed objects. At that time some other remembered set items may have caused some of the postponed objects to be moved somewhere sensible. Cheney scanning the moved objects may lead to even more objects of the postponed set being evacuated to places where they are referenced from.

[1]The first data slot of a `VMObject` is reused for the forward pointer (see section 6.7).

6.8.2. Stack Scanning

As discussed above, the stack scanning is done after the previous phases because references from the stack to either the introductory space or the from car do not really give us a good clue as to where to put the objects referenced (we cannot put them in the stack space as some early prototype attempted). Our interpretation of this is that it must be the closest we get to an ideal situation for creating new trains as objects that seem to be referenced only from the stacks include the objects that are “close to root objects”. It is still an unsolved mystery how often a new train should be created. We have experimented with two different schemes. The first one had a maximum limit of one new train for each scavenge, but the train was created immediately if there was just one object referenced from a stack that needed to be moved. As we shall see later in the experiments, this has the drawback of allocating a lot of trains with one car that is not very full. This motivated reuse of these new trains, so an extra constraint was set demanding that the car was filled above some threshold before a new train can be build.

An idea for future improvement is to do a Cheney scan before objects are evacuated from the stacks. This could potentially eliminate some evacuations of objects referenced from the stacks.

While the stacks are being scanned, the references are also checked for a reference to an object in any of the other cars within the from train. Unfortunately this operation is not cheap as each pointer needs to be checked for membership in every car in the train until one has been identified. In particular with long garbage trains and high stacks this operation is expensive as $stackItems \times (fromTrainCars - 1)$ membership checks are required to conclude that the rest of the cars in the train are not referenced from the stacks. This information is saved for later when reclaiming takes place.

After the stacks have been scanned, another Cheney scan is performed to update the moved objects and move and update their referenced objects.

6.8.3. Reclaiming Memory

In some cases it will now be possible to reclaim the entire from train. This is only possible when there are no external references to it. It has already been checked whether there are references from the stacks. If the external remembered sets are also empty, it can be concluded that the from train is reclaimable, otherwise only the from car is. In the latter case internally referenced objects should still be rescued to the from train using a scan of the internal remembered set on the from car followed by yet another Cheney scan. After the space in question has been reclaimed, the introductory space is reclaimed too, and the scavenge cycle is over.

6.9. The Write Barrier

The job of the write barrier is to do the actual pointer write and update the appropriate remembered set if necessary. In figure 6.15 the write barrier is shown with `updateVMReference()` inlined. We will now explain the interesting lines of this code.

```
void setVMReference(void **refAdr, void *target) {

  //set reference
  *refAdr = target;
                                                                5
  //update remembered sets
  CarTrain_t crTrSrc = trainGeneration.getCarTrain(refAdr);
  CarTrain_t crTrTrg = trainGeneration.getCarTrain(target);

  if(crTrSrc > crTrTrg) {                                      10
    if(crTrSrc.getTrain() == crTrTrg.getTrain()) {
      //internal reference
      (Car_t::getCar(target))->addIntRememberReference(refAdr);
    } else {
      //external reference                                     15
      if(introSpace.member(target)) {
        introSpace.addRememberReference(refAdr);
      } else {
        (Car_t::getCar(target))->addExtRememberReference(refAdr);
      }                                                        20
    }
  }
}
```

Figure 6.15.: *The write barrier implementation*

The parameter `refAdr` is the address of the reference that will be updated to point to `target`. The updating of the remembered sets and the tests involved in this are the most interesting aspect of this piece of code. In lines 7-8 the ordering of the reference and the target is retrieved and in line 10 these are compared. Only if the target is of higher order than the reference is it necessary to update remembered sets, and reference updates all inside either the introductory space or the same car are also filtered out here since their cars and trains would be equal. Line 11 identifies reference updates that are internal to a train using `getTrain()`. In line 16 we distinguish between external references where the target is in the introductory space and the train space, and update the appropriate remembered set. It is important to note that we can omit checks for references to root space because these are not permitted as object descriptor marked references, otherwise line 19 would have to include a check for target not being a member of root space.

It is important to note that according to [HMS92] many pointer writes seem to occur during initialization. If this is the case, most of these objects would be situated in the introductory space, and the write barrier will then already return in line 10.

With this implementation only the remember reference updates are not constant time. Remembered set updates have a worst case insertion time of $O(n)$ where n is the number of elements of the hash set when the hash set is resized, but in most cases the remembered sets have an insertion time of $O(1)$.

6.10. Interface

The memory management component should be as loosely coupled to the rest of the system as possible as it is of great advantage not having to know the details of the garbage collector when implementing the code executing part. For this reason we define functions that are independent of the specific collection algorithm and header information in objects used in the garbage collector. In this section we present the classes and methods the mutator needs to know and use in order for the memory management to work as intended.

6.10.1. Initiating Garbage Collection

Garbage collection is initiated by invoking `garbageCollect()` on the global instance of the `GarbageCollector` class. It is important to determine when this method should be invoked, whose responsibility it is, and how often it should be invoked.

One way to go is to create a “magic `alloc()`” that always allocates memory but sometimes also invokes `garbageCollect()`. This is nice because the allocation call is required anyway, and no extra calls are needed for doing the garbage collection. We have chosen a slightly different strategy where the memory manager is passed a function pointer to register the mutator. The memory manager calls this function to signal when it would like to garbage collect at the next safe point. It is up to the mutator to determine when this safe point has been reached. The garbage collector will signal the mutator when the introductory space has been filled up so the garbage collection frequency is basically determined by the size of the introductory space. New objects are allocated in the train space until garbage collection has been performed.

The reason for our unusual strategy in this respect is that it gives more freedom to the mutator. During our implementation in the last semester we discovered what we called the this-problem; when an object referred from the `C++` stack was moved, its `this` pointer would no longer be valid when the object was reentered (see chapter 5). The this-problem can be generalized to garbage collecting while not having the complete root set which is bound to cause trouble. This motivated our current scheme where garbage collection can only occur at specific places controlled by the mutator. If garbage collection could occur every place where memory was allocated directly or indirectly during instruction interpretations, we were afraid that we would not be in a consistent state sometimes e.g., by allocating two or more objects and afterwards setting up pointers between them and finally setting a pointer to keep them alive.

6.10.2. Reference Placement in Objects

Before memory can be allocated for an object, an *object descriptor* must be created or found (they can be shared if they are not modified during execution). Object descriptors are placed outside the heap memory managed by the garbage collector and can be allocated using the default `new` operator. The constructor takes a `length` parameter in bytes and a `mask-value` which can be used for initializing the descriptor to having references at certain points. The purpose of the `mask-value` is to allow easy setup of objects which have no references in which case it should be set to 0, or objects which only have references in which case it should be set to the number which is all ones in binary representation. In the last case the excess reference indicators (see figure 6.16) can safely be ignored as no reference beyond `length` (converted to references) are considered by the garbage collector.

Object descriptors can be further customized after initialization by using the methods in the `ObjectDescriptor` class: `setReferenceAt(position)` and `clearReferenceAt(position)`, but care must be taken if the object descriptor is shared between objects, as changes to the object descriptor will affect all objects sharing the object descriptor. This feature of altering an object descriptor while executing is not exploited by our current mutator.

6.10.3. Memory Allocation

Memory allocation is performed using a number of macros: `allocateVM()` for allocating normal collected objects, `allocateRootVM_NT()` for allocating non-traced root objects like integer stacks, and finally `allocateRootVM_T()` for allocating traced root objects. Each one of these macros takes an object descriptor reference and returns a pointer to the allocated memory.

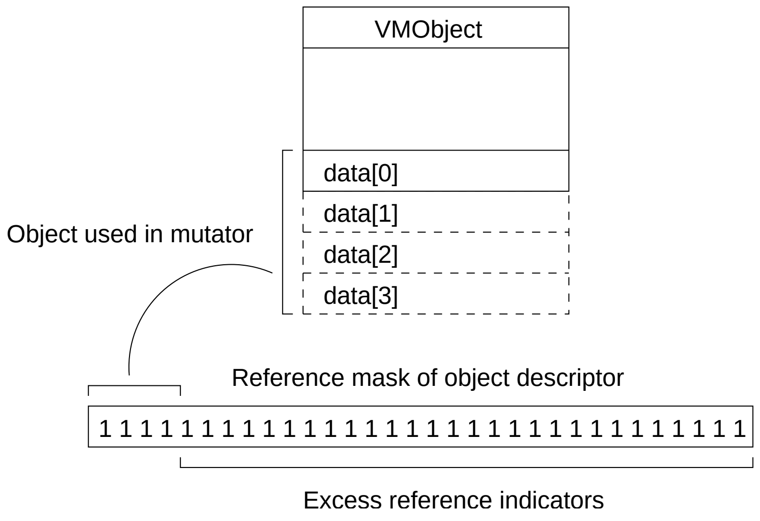


Figure 6.16.: *Illustration of the reference mask of the object descriptor of an object used by the mutator with a size of four references. The object descriptor indicates using four bits that all the fields in the object are references, and the rest of its bits are excess reference indicators*

6.10.4. Setting and Changing References

In order to implement some garbage collection algorithms including ours, it is necessary to be able to trap pointer updates (see section 4.2). To handle this the write barrier function: `setVMReference(object, reference-location, new-reference-value)` should be used. `object` is the object embedded in a `VMObject` that the pointer should be modified in. This is no longer used by the new garbage collector and is only there for compatibility with the old garbage collector. This has no influence on performance when `setVMReference()` is implemented as a macro. `reference-location` is the absolute location of the reference in that object, and `new-reference-value` is the new value that should be placed into the reference-location.

It is vital to understand that this function should only be used when handling references from objects embedded in `VMObject` instances to other objects embedded in `VMObject` instances – in other words references which should be handled by the garbage collector must be modified with this function, and others must not. A notable exception to this is references to root objects. These must not be set with `setVMReference()` and there is no need to since these objects are never moved.

6.11. Debugging Memory Management Systems

As stated in [Hud00], debugging memory management systems is a difficult task, and we could not agree more. We have approached this task in multiple ways. The first was to print a lot of ad hoc debugging information during garbage collection. The amount of information produced this way quickly increases and becomes difficult to comprehend so spotting problems in the implementation gets even more difficult. A part of this debugging information is only relevant in one context, while other parts are only relevant in other contexts, while even other contexts may require some of the information from both of the former two. Commenting out printing of debug information is one way to adopt the output to the problem in question, but as the size of the code increases this gets impractical. It is also a tedious task to comment out all the print statements when code is “ready for release” and reenabling them when a new bug is introduced later.

6.11.1. Control of Debug Printing

To solve these issues a small print control system was designed and implemented. This system is based on macros in the `C++` preprocessor, enabling us to redefine them to do nothing when a “release” is build and thus causing no performance overhead in such versions.

When development versions are built, the first thing one must do is to set up what categories of debugging information should be printed and which should not. This is done using the `DB_ENABLE(<category>)`, `DB_ENABLE_ALL`, and `DB_DISABLE(<category>)` macros. An example of such a category could be “gc_remset” to indicate that it has something to do with the garbage collector and the remembered set. Each category is assigned a unique forth running number used as index in a global table. This table remembers whether message categories should be printed or not and the above macros only modifies it.

Whenever something is to be printed in this system, one must use the macro `DB_OUT(<category>, <what-to-print>)`. Only one `<category>` can be specified, so one will have to choose carefully, but this has not proven to be a severe restriction. More importantly `<what-to-print>` is not restricted this way; strings can be composed with the standard `C++` $<<$ operator. The macro basically wraps the print statement in an if-construct where the condition expression consults the category-table with the given category index.

In addition to this control of the output, indentation of the output also proved to increase its readability.

Another thing that makes it easier to realize where in the byte-code the virtual machine is executing is the possibility to pretty print the instruction lists of gbc-main-parts. This includes nice indented printing of multi-line instructions and attribute initialization lists.

6.11.2. Heap Consistency

After having worked with the implementation of the memory management system for awhile, we discovered that a common bug symptom was an illegal pointer somewhere in the heap. Our first reaction to this was to implement methods for printing out the entire heap and the associated remembered sets. This was very powerful as long as the heap was small and populated artificially (no real interpretation was going on).

When our virtual machine component was attached instead, it became impractical to trace bugs this way so we implemented methods for doing the work for us. The first version collected all valid reference values by scanning the entire heap. This was followed by an additional scan which checked every reference in every object of the heap against the valid references. This version had the flaw that it did not detect inconsistencies in the remembered sets, which implied that an error in the remembered set update would only be discovered when the piece of memory with the corrupted remembered set was scavenged. Only when checks of the remembered set for each reference were added, the heap consistency tests became really good at stopping at the right moment in execution. We verified the remembered sets by checking the existence of remember-references, i.e., some remember-references must be present while others must not according to the ordering of trains and cars. Both the internal- and external remembered sets are checked this way.

This check of the heap consistency is quite expensive but certainly much quicker than doing the work manually. An unpredicted advantage of heap consistency checks is that they can also be used in the virtual machine component for debugging instruction implementations. Errors like setting references without using the write barrier and wrong values in references were easily tracked this way. Since the check is quite expensive, it cannot be run between each instruction at reasonable speed. Counting the instructions and doing a check each, say, every 1000th instruction, gives a rough idea of the place in execution where something is failing. gbvm can then be instrumented to do checks between every instruction after e.g., the 151000th instruction. This typically identifies the exact instruction that is failing. However, there are instruction bugs that cannot be identified this way either because they are an interrelation between two or more instructions or because the error is of a semantic nature.

Running the check each time the virtual machine switches between interpreting byte code and garbage collecting and the other way around, also places the responsibility of a bug quickly.

6.11.3. Data Display Debugger

To debug our system we also used the GNU Project application: Data Display Debugger /v 3.x (`ddd`) [Pro01]. `ddd` is a graphical front-end for command-line debuggers such as The GNU Debugger (`gdb`). When debugging the virtual machine we used the debug prints to trace bug even when they resulted in segmentation faults (what they usually did). This was hard work but it also resulted in a very thorough code walk-through. When later we became aware of the qualities of `ddd`, it was used to back-trace from segmentation faults. This made debugging much easier as long as the bug was not placed in a macro, since `gdb` and therefore `ddd` cannot back-trace into macros. In both the virtual machine and memory management component we use macros often and this somewhat decreases the application of `ddd` since it is not capable of expanding macros (as far as we know).

6.11.4. Rational Purify

Rational Purify [Cor01] is a commercial but very efficient application debugger. It excels in detection of memory leaks and memory accesses outside allocated ranges. We used a 14-days evaluation copy of this product to identify a number of bugs in our system. Purify made us e.g., realize that a problem with an unstable version of our system was caused by main-part id char-arrays being allocated one byte too short. The source of this problem would have been difficult to identify without a tool like this. Commonly nothing seems to happen immediately after a byte just outside an allocated range is written and sometime, perhaps much later, the program fails in a place where it never did before for no apparent reason, and when trying to identify the problem, it suddenly vanishes because the debugging code needs memory allocations too. This tool is capable of pinpointing both the location of the allocation and the illegal read/write.

The only drawback to this tool that we are aware of is its inability to do bounds checking inside the blocks allocated with e.g., `malloc()` or `memalign()` since it is not aware of types. It only considers allocated versus not allocated. As we allocate large blocks for our heap and manage these ourselves, the bounds checking is not operational in the majority of the memory manager. It does e.g., not seem to have any problem with the way we put new objects in the end of and outside cars, although the array size of one might indicate this was a terrible mistake. In this case this is fortunate since this is intentional, but it also means that it cannot detect if `VMObjects` have too little memory allocated for them.

6.11.5. Discussion

Our experience is that when one starts to rely on that the debugging system will report newly introduced bugs, one gets the courage to experiment with the system. Its like when a circus acrobat has a safety net he tends to have the courage to try more crazy stunts than without the safety net. In particular when the system approaches a stable state, and one starts to experiment with minor changes, it is beneficial to be able to render probable that new changes do not break the system or otherwise quickly identify problems introduced.

The debug printing system of gbvm is one of the generic parts that will probably be worth using in future projects. The heap consistency check is more specific for memory management systems and thus not as widely applicable, but it has been a very powerful tool in tracking bugs. The `ddd/gdb` combination and Purify® are general debugging tools that are useful in most application development.

7. Experiments

In this chapter we present some performance criteria that a memory management system should meet. Then we analyze how these criteria relate to our memory management system and how they can be measured. Before presenting our experiments we outline the general methodology used. The chapter ends with a section that discusses the results and conclusions drawn from the experiments.

7.1. Performance Criteria

A garbage collector must be a compromise between the below criteria.

7.1.1. Garbage Collector Time Efficiency

How time efficient a garbage collector is can be measured as the time it uses to garbage collect. This can be measured as the total, mean, and maximum disruption time.

But how do one conclude that a garbage collector is nondisruptive? Disruptiveness is hard to quantify, since it depends on the requirements to the program being executed. It relates to the real time demands for a system. A real time system can be a hard real time system or a soft real time system. The difference lies in the type of deadlines the system must obey. Whereas a hard deadline must be met, a soft deadline can be missed from time to time [Sta97]. If hard deadlines should be met, it must be ensured that a garbage collector has a well defined worst case maximum garbage collection time and frequency.

With typical train algorithm garbage collectors it is impossible to meet hard real time demands because there is no special handling of popular objects. Still, low mean time disruptiveness is an important performance criteria in applications where responsiveness is important, e.g., interactive applications.

7.1.2. Mutator Time Overhead Related to Garbage Collection

It is vital that the memory management component has little effect on the time used by the mutator at run-time. A memory management system *can* affect the time used at run-time in at least two ways. The overhead of the write barrier to track interesting pointers, and the overhead introduced when new objects are allocated.

7.1.3. Garbage Collector Space Efficiency

It is important that the garbage collector is space efficient, i.e., it utilizes its allocated memory well. A garbage collector can waste space in several ways. It can allocate blocks without using them at all. It can also waste space inside used blocks as both space allocated by dead objects and space not allocated by objects.

There are also other sources of space waste in garbage collectors. In our case, we waste space used by `VMObject` headers, `Car` headers, the train table, the car-ordering table, and the remembered sets, to mention some. The most interesting, in our opinion, is the space unused in used memory blocks (cars or introductory space) and the amount of dead objects in the heap.

7.2. Possibilities for Analyzing Our System

In this section we analyze the possibilities for analyzing our system.

7.2.1. Introductory Space Size vs. Car Size

The sizes of the introductory space and the cars are believed to have a significant impact on the performance of the virtual machine both with regard to time and space consumption. The size of the introductory space effectively determines the garbage collection frequency since garbage collection is started shortly after the introductory space has been filled. This is important since a higher frequency makes the garbage collector better at keeping up with the mutator creating garbage if the size of the cars is kept constant.

Since the majority of objects are assumed to have a short life, increasing the size of the introductory space should improve performance as a smaller part of the introductory space will have to be copied. This is expected to reduce the overall time used for garbage collection, but it is difficult to observe this alone. When you increase the introductory space size, the number of cars scavenged will be reduced too, thus further reducing the garbage collection time used. When the assumption of a short average lifetime of a new objects does not hold, a large introductory space may impact the disruptiveness during garbage collection since a lot of objects will have to be copied. With gbeta it is common that many objects do die young since method calls are also objects. If these activation-record like objects were placed on a stack instead, the garbage collection overhead due to method calls would be decreased. A large introductory space and/or car size should reduce the write barrier overhead since it is cheaper to create references between objects inside the same block[1]. It should also increase the probability of objects referencing each other being placed in the same block. The reason is that there is a greater chance that a sub object graph fits inside the block when the train algorithm evacuates objects. Another reason is that there are more objects inside the block so they coincidentally happen to be in the same block. The better locality makes the garbage collector better at reclaiming garbage structures quickly [SaCL00]. Better locality also makes the CPU execute faster since the number of page faults is decreased and the caching behaviour of the CPU is improved too.

7.2.2. Write Barrier Performance

The performance of the write barrier is also expected to have a significant impact on the overall performance of the system. It would be interesting to know if the write barrier consumes so much time that a redesign is required for acceptable performance.

7.2.3. Allocation Performance

Since gbeta is very allocation intensive, we suspect the performance of the allocation routine is important. As with the write barrier performance it would also be interesting to know if our allocation routine is unacceptably slow.

7.2.4. Altering The Algorithms

The train algorithm allows some freedom in various aspects, especially with regard to the destination when objects have to be moved, and the policy for creating new trains and cars. These policies are based on

[1] by *block* we mean `Car` or `IntrodoctorySpace`, not `MemoryBlock`

somewhat qualified guesses, but we would like to change that with systematic experiments. Depending on how thoroughly the effect of changes to the algorithm is measured, it quickly becomes tiresome to conduct the experiments and evaluate the results since each new independent change doubles the number of test runs required when all combinations are tried. This exponential growth discourages us from doing a lot of these experiments although they are very interesting. Only when we suspect the presence of a problem will such experiments be justified.

The experimentation framework developed with this thesis is relatively user-friendly, so it is possible for memory management researchers or others to continue at the point where we are now. It is possible to experiment with parameters such as car and introductory space size, and with some source code knowledge, it would also be possible to experiment with algorithm changes such as changing the write barrier algorithm or changing the policy used to create new trains.

7.2.5. Profiling

Profiling of a running executable is a way to reveal which parts consume large fractions of the total execution time. The compiler we used, `g++`, has builtin support for profiling when given an extra argument. This profiling is based on function call-counters and periodic sampling of where in the code the processor is executing. The sample period is 0.01 second and not user definable. When a program only runs a short period of time, this gives a high uncertainty because only a few samples will be taken. For this reason it is possible to combine the output of many runs to get a better certainty. For more information on the profiler we have used, `gprof`, see [GKM82].

7.2.6. Overall Time Efficiency

Our virtual machine must be fast overall since this is vital for any virtual machine. An efficient gbeta virtual machine could also spread out the use of gbeta. Although, this is not the focus of this thesis it is interesting to see how competitive gbvm is.

7.2.7. Choice of Analyses

As our hypothesis states, we focus on memory management and especially what happens if some of the parameters and policies in the train algorithm are modified. We have therefore chosen to do the following experiments: measuring space performance when altering the car and introductory space size (see section 7.4), time performance when altering the car and introductory space size (see section 7.5), experiments with the policy used to add new trains (see section 7.6), measurements of the time used in the write barrier (see section 7.8), profiling the system to identify time fraction used by different parts of gbvm (see section 7.9), a speed comparison of the existing gbeta executing systems (see section 7.7), and finally a little experiment where we compare gbvm with the Java virtual machine (see section 7.10).

7.3. Methodology

In this section we present the general methodology used in the following experiments. That is, we present the hardware used, the test programs used, and the general presumptions and uncertainty connected with the experiments.

7.3.1. Hardware and Software Used

The experiments that involved time measurement were all run on the following hardware (see appendix B for more details):

CPU Pentium 133 MHz (with f00f_bug) – 53.04 bogomips.

Memory 64MB (60ns EDO-ram)

PCI-devices 10/100Mbit Ethernet controller, ATI Mach64 VT VGA-controller

Operating System Linux 2.2.14-5.0 (Redhat 6.2)

Compiler GNU g++ 2.95.3 with i586 target CPU

7.3.2. Test Programs

Since gbeta is neither a very well known nor a widely used language, and since we do not support concurrency and repititions, we only have two sources of test programs: The programs used to test gbeta (implemented by Erik Ernst) and programs we have implemented ourself. This is a problem since [ZG92] has pointed out that the only way to really test the performance of a garbage collector is to use real-life programs as test programs [ZG92]. Even though this is a problem, it is common to use self-implemented programs to evaluate memory managers [HMS92]. This gets problematic when one implements test programs to demonstrate the advantage of specific features in a given implementation instead of checking whether it really is an advantage in real programs. Moreover, the test programs, we implemented, tend to be a lot shorter and have the same behaviour repeated for a long time which is probably not representative for the majority of real programs. In effect this means that the reliability of the results is not as great, as we would like because a new set of test programs could change the results considerably.

We have implemented the majority of the following test programs. In appendix A all but `tst-norep-3.gbc` and `tst-norep3BO.gbc` are listed.

`tst-norep3.gbc` A test program originating from the development of BETA. We have modified it to remove or recode parts using repetitions, concurrency, or other unsupported instructions. We have added a loop to the main do-part to make the program run the same code ten times and removed a section where big object handling was tested.

The purpose of this program is to bound test a given gbeta execution system. It consists of fifteen test methods that test everything from relational operations to virtual attributes.

`tst-norep3BO.gbc` The same as `tst-norep3.gbc` but including the test where big object handling is tested. The big object test proved to have a significant impact on the behaviour of the memory manager, so we included both the version with and without this test.

`allocator.gbc` The purpose of the program is to allocate several small objects and invoke them, and thereby create a lot of short lived objects.

`cruncher3.gbc` This program is both allocation and computational relatively heavy. It constructs a tree breadth-first, and after the tree has been constructed it iterates the entire tree three times using depth-first search.

`constAlloc.gbc` Does the same as `cruncher3.gbc` but during the depth-first search, a sub tree is created and a specific node in the original tree is repeatedly replaced by this sub tree. This results in a program that constantly allocates the space used by the subtree.

`dfstree.gbc` Does the same as `cruncher3.gbc` except that the tree is constructed depth-first. This is expected to infer better locality of reference than a breadth-first creation.

`cruncher3JAVA.gbc` A simplified version of `cruncher3.gbc` made to enable a more even comparison between the Java virtual machine and gbvm.

`cruncher3.java` A program that does almost the same as `cruncher3JAVA.gbc`, but implemented in Java.

`derive.gbc` This computationally heavy, but allocation and garbage light program computes a number of product sums 50000 times.

`simple.gbc` Calculates 1 + 1 a thousand times. This program was constructed to increase the number of speed comparisons between gbetai [JJW01] and gbvm.

In the following tests we have tried to select the source files that gave the most distinct and interesting results. Especially `cruncher3.gbc` and `dfstree.gbc` yielded very similar results in some of the tests, so only the `dfstree.gbc` was used. Also, `derive.gbc` and `simple.gbc` were excluded from the garbage collection tests because the jobs were not sufficiently allocation intensive and the results were very similar.

7.3.3. Presumptions and Uncertainty

We are not aware of any comparable setup and have not implemented other garbage collection algorithms to compare with this one e.g., one with a copy collector in the introductory space. This is not a problem since the purpose of these experiments is to try out some new ideas and see what happens. Besides that, our virtual machine is executing gbeta – a not so widespread language – so we only have any one gbeta executing garbage collecting environment to compare with, namely gbetai [JJW01].

Test Program Uncertainty

As discussed in section 7.3.2, the choice of test programs is an important source of uncertainty.

Hardware and Setup Uncertainty

To reduce the uncertainty originating from external factors while executing tests involving time measuring, our test machine was running in single user mode with only essential other processes running. Also the networking and swap space was disabled.

Since the resolution of the `time` command is only 0.01 second, it is important that the processor of the test hardware is not too fast because this gives better statistical measuring certainty with the same number of observations.

Size Ranges of Introductory Space and Cars

In some of the following tests, we have chosen to use a range of sizes for the cars and introductory space:

Car sizes 4KB, 8KB, 16KB, ... , 256KB

Introductory space sizes 4KB, 8KB, 16KB, ... , 1024KB

We make no claim that interesting sizes cannot exist outside these ranges, but it was impractical to include more sizes as the total execution time of the tests wold be even longer, and some setups would use more memory than available on our test machine. The reason for the lower bound is that the lower of these sets the limit of the maximum object size. We found it unacceptable not to be able to handle objects bigger than

4KB. The larger bound was found by looking at the setup in [GS93, p92] and doubling this twice. The reason why the introductory space is doubled four times (to 1MB) instead of two is that we suspect that the introductory space would have to be larger than a car because of the common short lifetime of objects.

Test runs with large introductory spaces suffer from very few measurements for some of the test runs and thus have a high uncertainty. Increasing the certainty of these would require us to increase the size of the test programs (e.g., in number of iterations), but this has a very high time cost with some combination of the smaller introductory spaces and large car sizes. This is also a major reason why larger cars and introductory spaces have not been included.

The choice of the introductory space and car size somewhat depends on the present hardware performance. The chosen ranges might be too small if similar experiments were done in the future, and correspondingly too large, if the experiments had been done e.g., ten years ago. Since our test hardware is both slower and have smaller main memory than todays common hardware, we might have chosen too small ranges. Again, our generic experimentation framework makes it possible to easily conduct the same experiments on more up-to-date hardware with other ranges and larger test programs. Our hope is that the results will be independent of the actual sizes used and show a tendency generalizable to larger sizes.

7.3.4. Structure of Experiments

In each of the experiments in the subsequent sections we explain the purpose of the experiment, present the method used and which subset of the test programs is used, present the results and finally evaluate the experiment.

7.4. Space Utilization in Cars and Introductory Space

The space waste on the heap relates to the size of cars and of the introductory space. The purpose of this experiment is to find an optimal car size and introductory space size with respect to space waste.

Space waste is the amount of memory allocated by gbvm for `VMObjects` which is not used. Space waste can be divided into two kinds: Space occupied by dead objects and space which is not allocated at the end of a car or the introductory space, i.e., *unused space*. These two kinds of space waste are not equally bad. The dead objects are more difficult to remove from the system than unused bytes because dead objects can reference each other across car/introductory space boundaries and thus keep each other alive for a while causing collecting overhead. Unused bytes on the other hand do not cause additional processor overhead and even reduce the disruption time as less heap is scavenged.

We have decided against including the memory used by the remembered sets and stack space in the measurements; only the train space heap and introductory space heap are being observed. If the other sources of memory consumption were included, it would have been more difficult to say anything about the cause of a given problem. The stack space is currently uninteresting as the memory consumption is constant. Also, the remembered sets is a science in its own right and deserves a more thorough investigation.

7.4.1. Method

To measure the amount of live, dead, and unused space in the introductory space and the train space, the memory manager is instrumented to mark all the live objects like the mark phase of the mark and sweep algorithm. Instead of the sweeping phase, the bytes in the three categories are counted and the objects are unmarked. Extra per-object overhead is avoided by using the forward flag for the object marking. This data collection is run before and after each garbage collection, and the numbers are output separately for the introductory space and the train space. This allows us to investigate the train space and introductory space separately and combined.

Since the result of this test is not dependent on hardware speed it has been conducted on faster Linux hardware.

Behaviour over Allocation-time

To get an intuitive understanding of the behaviour of the test programs and the handling from the garbage collector, the live, dead, and unused bytes are shown in 2D graphs with 4KB introductory space and 4KB car size, and the new train creation policy is used (which is the subject of section 7.6). The choice of 4KB introductory space size gives the finest granularity in the graphs. With other introductory space and car sizes, the graphs changes. The live graph is only dependent on the introductory space size, and the two other graphs generally vary with both introductory space size and car size.

The free variable in these graphs is what we would call *allocation-time* – two samples are taken each time the introductory space is filled; the first sample just before the garbage collection and the second just after. The unit on the first axis contains two such samples within each increment of one, thus the unit shown is the number of allocations of about the introductory space size.

Only the train generation is used the in dead and unused parts of these graphs because the introductory space introduces a lot of "noise". Before garbage collection the introductory space is filled with live and dead objects, hardly any space is unused, but after the collection, all of the introductory space is unused. This does not affect the number of live bytes since these are moved to the train space, but the dead and unused graphs jump a lot because of this, particularly with large introductory spaces with few live objects.

Effect of Introductory Space Size and Car Sizes

For the introductory space alone, the train space alone, and the two spaces combined, we (for each) generate three 3D graphs showing the average amount of live, dead, and unused space against the two free variables: introductory space size and car size. This produces a total of nine 3D graphs for each test file. As the live, dead, and unused graphs show fractions of the whole space under investigation, adding the three graphs point by point would thus yield a plane with the value 1 everywhere. It also means that the graphs affect each other as a growth in a region of e.g., the unused fraction will induce a smaller fraction in the two other graphs in that region if these are unaltered. This perhaps makes the graphs harder to read as it is more difficult to observe in which of the fractions that is inducing the change. The absolute amount of live objects is, however, somewhat constant (see section 7.4.2), so a drop in this graph usually indicates a rise in one of the two others.

The complete set of 3D graphs can be seen in appendix C. It should be noted that the first garbage collection is not included in the calculations since the system had not had any chance of stabilizing itself; the train space is always wasting a large fraction of its space before it has had a chance to be filled.

7.4.2. Uncertainty

It is interesting to note that some virtual memory systems (the one in Linux at least) will not allocate the memory pages a process has requested before the pages concerned are actually accessed. This means that the unused space, which we calculate as the memory from the free pointer of a car to the end of the car, may not actually be allocated. This again means that our measurements may not be accurate when new cars are allocated, but after the car has been reused a couple of times all memory will be allocated for it.

There is also a uncertainty related to how often we measure the amount of live, dead, and unused bytes. If we measured this between each executed instruction, all our measurements would be directly comparable since we would then be sure that the number of bytes measured would be accurate. As this requires too much processor time to do this, we do the sampling before and after each garbage collection. This means we get fewer measurements with larger introductory spaces and thus our numbers are more uncertain as the size of this space rises. This is particularly bad since the number of garbage collections (and thus samples) is halved each time the introductory space size is doubled. What this all means is that the measurements with the same introductory space size are directly comparable (but not necessarily accurate) since the samples have been taken at the same time. These rows with the same introductory space size are only approximatively comparable with other rows having the another fixed introductory space size. If we had to redo this experiment in the future, we would consider eliminating the uneven sampling of the experiments with small and large introductory space sizes.

7.4.3. Results

All the 3D graphs presented in this subsection can be found in appendix C with both the old and the new train creation policy. Here we only use results from the new policy, though.

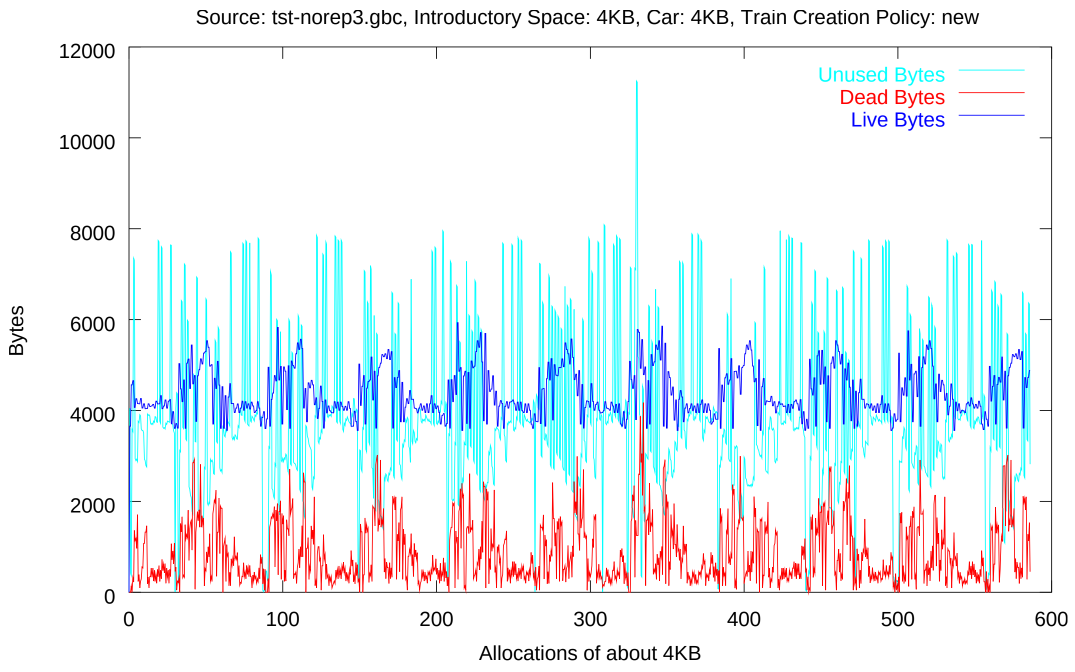


Figure 7.1.: *Memory behaviour of `test-norep3.gbc` with 4KB car and introductory space sizes.*

The figure 7.1 shows the allocation graph of `tstnorep3.gbc`. Although this program executes a lot of different instructions during the tests it is performing, it has very low memory requirements. Even with the smallest car size of 4KB all live data can almost be stored inside a single car.

The allocation profile for `allocator.gbc` shown in figure 7.2 displays low and very constant memory requirements. Adding the graphs gives a value very close to 4KB so this test typically only needs one car in the train space.

Figure 7.3 shows the allocation graph for `constAlloc.gbc`. This program initially allocates a large amount of memory of which some is later released. Towards the end, the memory requirements for the live data stabilize on about one third of the peak value. This graph shows the garbage collector having trouble collecting the dead objects.

The program `dfstree.gbc` in figure 7.4 shows an allocation curve constantly rising. The live bytes allocated peaks at about 1.9MB and relatively few dead bytes exist in the system. This is actually the

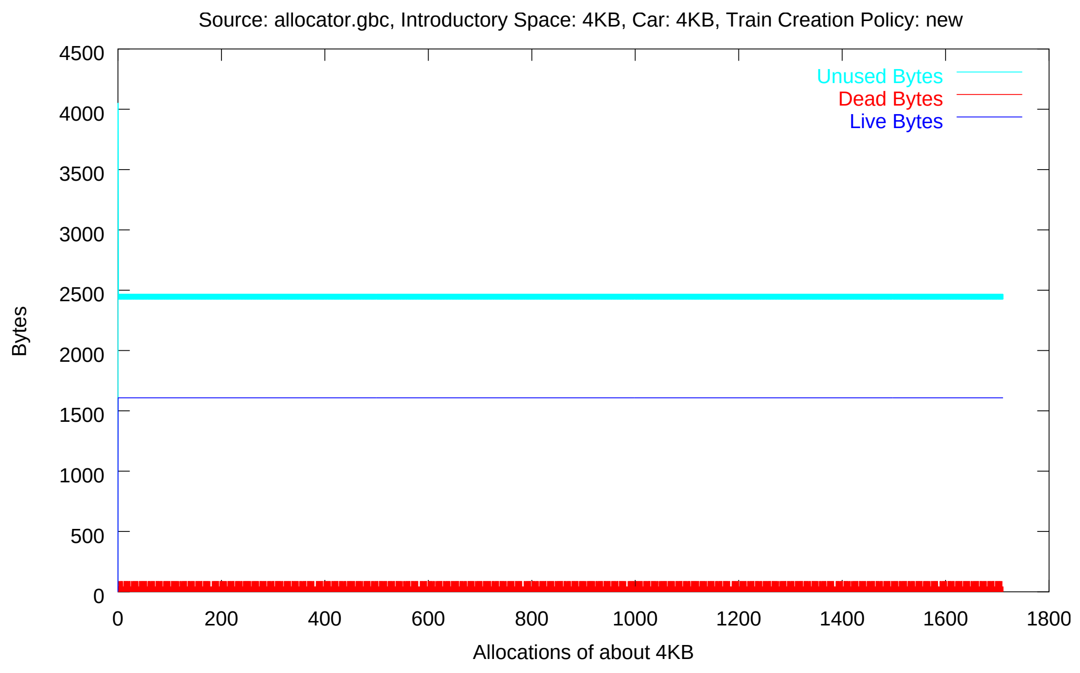


Figure 7.2.: *Memory behaviour of `allocator.gbc` with 4KB car and introductory space sizes.*

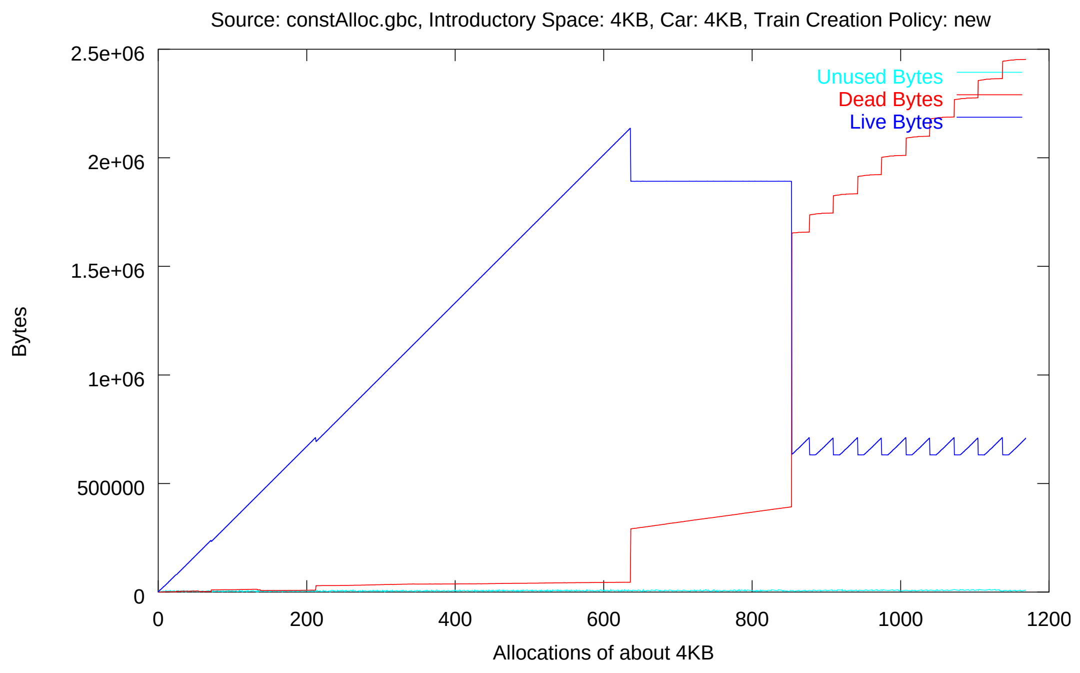


Figure 7.3.: *Memory behaviour of `constAlloc.gbc` with 4KB car and introductory space sizes.*

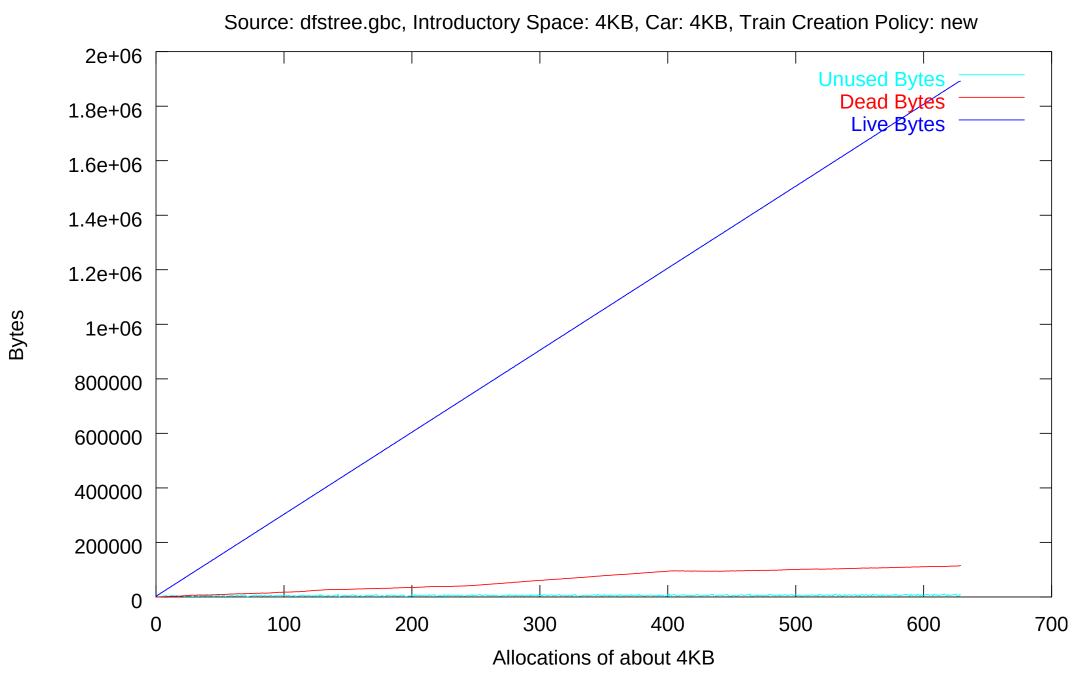


Figure 7.4.: *Memory behaviour of* `dfstree.gbc` *with 4KB car and introductory space sizes.*

example from the set of `dfstree.gbc` runs with different introductory space size and car size which has the highest dead bytes fraction.

In figure 7.5 the test `tst-norep3BO.gbc` includes the big object test in addition to the tests of `tst-norep3.gbc`. This additional part of the test periodically allocates a large chunk of memory which is quickly unreferenced. This allocation of more than 300KB is larger than the largest car size used in the tests (256KB) but smaller than the largest introductory space (1MB).

To show the effect of varying the introductory space and car sizes, we now show the results of taking each of them to their extreme as well as setting them both to 32KB as a compromise. This is all done with the test `tst-norep3BO.gbc` as this has a very visible effect on the execution of this source file.

First, increasing the introductory space to 1MB results in the graph shown in figure 7.6. The large introductory space filters out many of the allocation peaks in the live graph as well as a large part of the garbage. Compared to the situation in figure 7.5 the amount of dead bytes in the end has been decreased from about 1.9MB to about 1.1MB. It is unlikely that this system will be good at reclaiming the dead bytes in the train space, since the introductory space size dictates a low scavenge rate, and a structure of 300KB takes up at least 75 cars. If these cars are not lined up in the from train and “unpolluted” by live objects, it may take quite a few garbage collections before this structure will be reclaimed.

Increasing the car size instead of the introductory space size, changes this situation (see figure 7.7). Here the garbage collection rate is high and a larger part of the train space is scavenged which makes the garbage collector much better at keeping up. It is clear from the graph that the train space only contains between one and three cars.

Setting both the introductory space size and the car size to 32KB gives the result shown in figure 7.8. Compared to figure 7.5 this graph looks somewhat similar, but it is a little better at reclaiming dead bytes. This is probably the effect of the larger car size; the train space is divided into fewer cars and can thus be scavenged more effectively.

Looking at the 3D space utilization graphs for the train space only, the programs `tst-norep3.gbc`

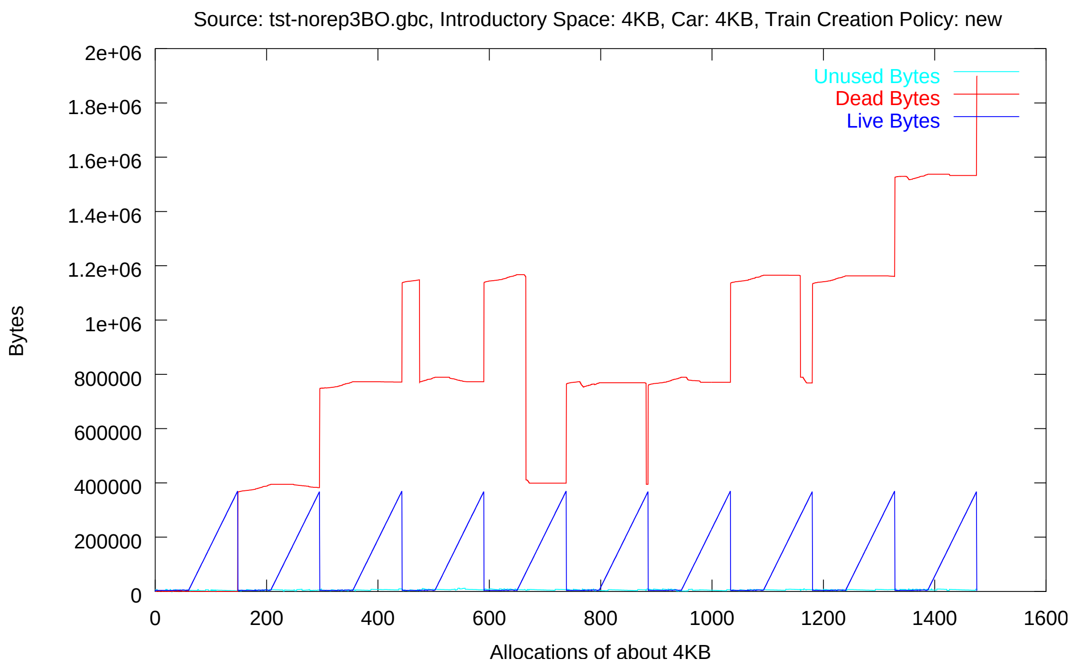


Figure 7.5.: *Memory behaviour of `tst-norep3BO.gbc` with 4KB car and introductory space sizes.*

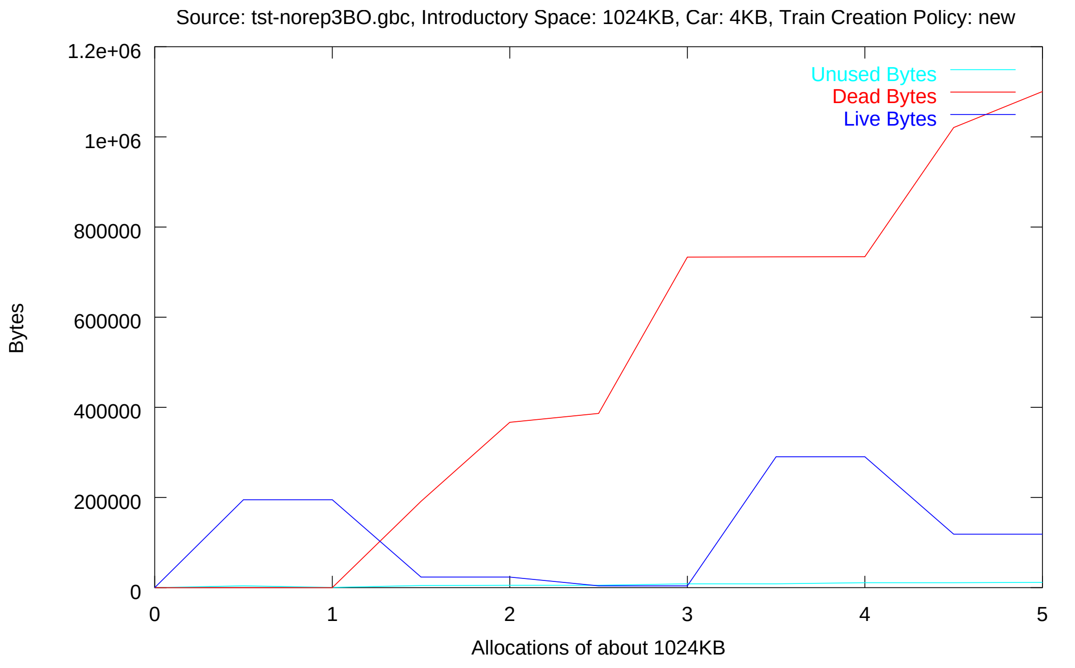


Figure 7.6.: *Memory behaviour of `tst-norep3BO.gbc` with large introductory space and small car size.*

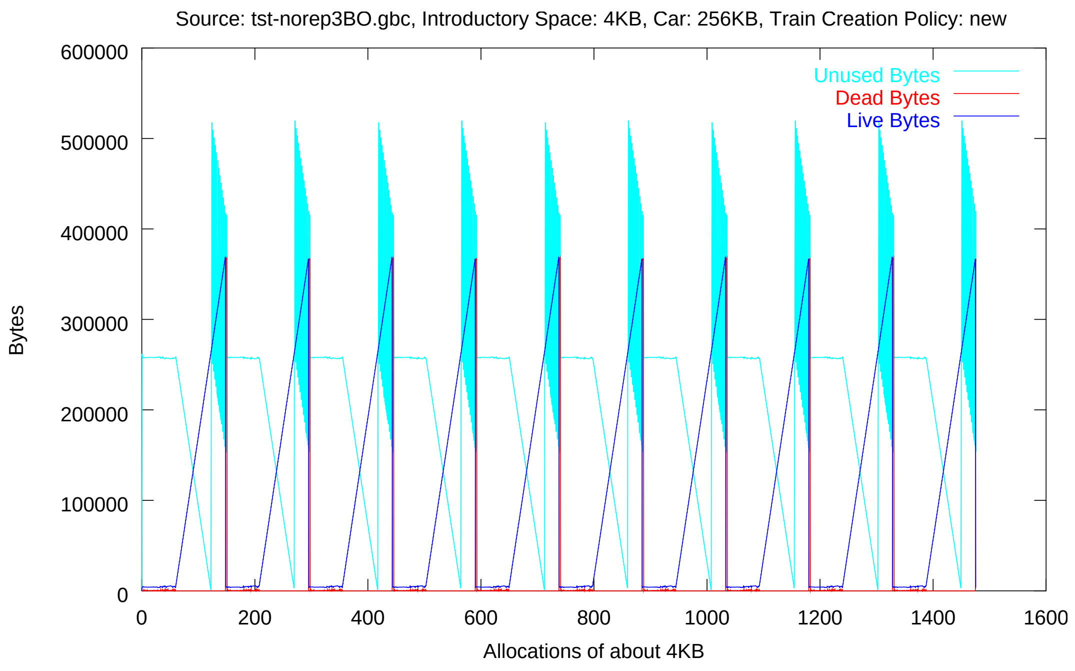


Figure 7.7.: *Memory behaviour of* `tst-norep3BO.gbc` *with small introductory space size and large car size.*

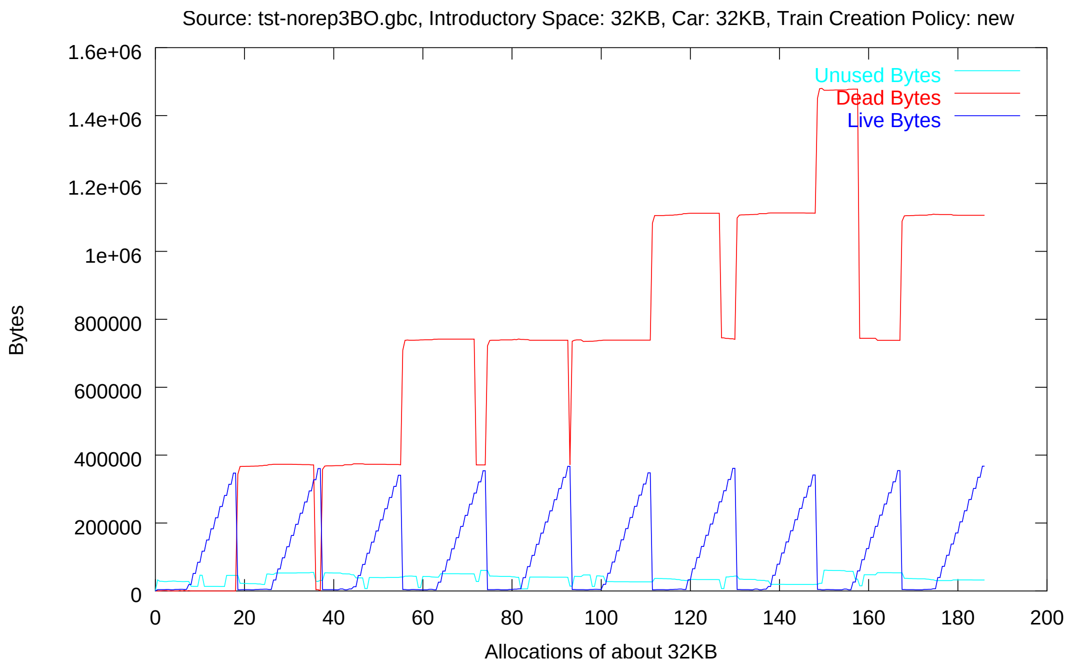


Figure 7.8.: *Memory behaviour of* `tst-norep3BO.gbc` *with 32KB introductory space size and 32KB car size.*

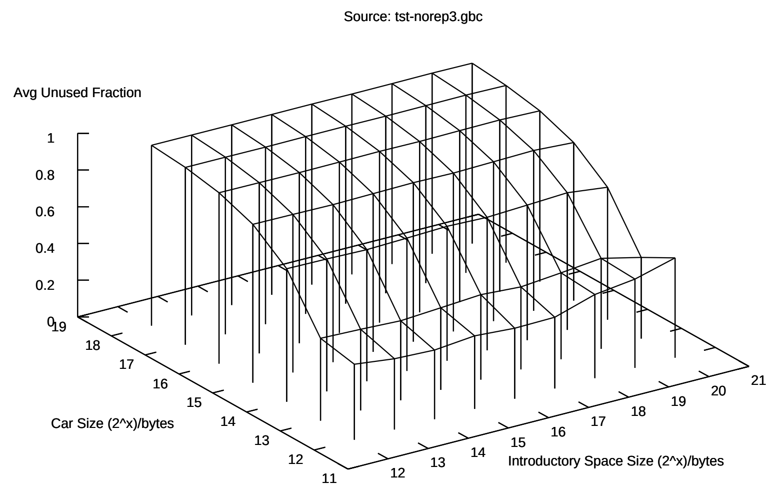


Figure 7.9.: *Average unused fraction in the train space executing* `tst-norep3.gbc`

and `allocator.gbc` seem very similar. When considering their allocation profiles in figures 7.1 and 7.2 this is not surprising. An interesting part of these graphs is the way the unused space is taking up an increasingly high fraction of the total amount of space as the size of the car rises while they are virtually independent of the size of the introductory space (see figure 7.9). The reason for the independence of the introductory space size is that very few live objects are introduced (see figure 7.10). The reason for the high amount of unused space with larger cars is that the car size becomes larger than the total number of live bytes and thus forces the unused fraction up.

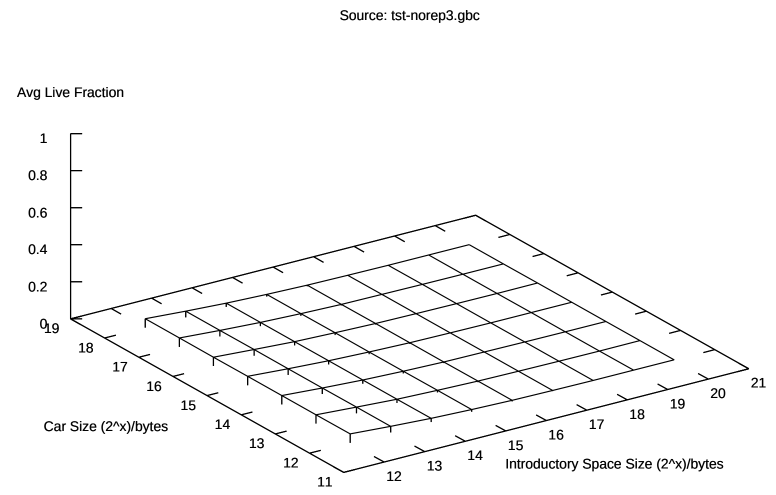


Figure 7.10.: *Average live fraction in the introductory space executing* `tst-norep3.gbc`

The live fraction of `tst-norep3BO.gbc` in figure 7.11 is one of the most peculiar graphs in the whole set. It shows that the live fraction is highest when the car size is larger than the introductory space size, but not too much larger! The drop towards the corner where the car size is large and the introductory space is small can be understood by looking at the two other graphs for this test series (figures 7.12 and 7.13). As discussed when the allocation profiles of `tst-norep3BO.gbc` were presented earlier in this subsection, the dead fraction clearly demonstrates the potential problem of garbage collecting too slowly when the introductory space is larger than the car size (see e.g., figure 7.6). This problem is vanishing as the corner in question is approached thus making room for a higher live graph in this corner. The drop in the live fraction towards the corner is caused by the rise of the unused fraction which again has to do with the large car size wasting space.

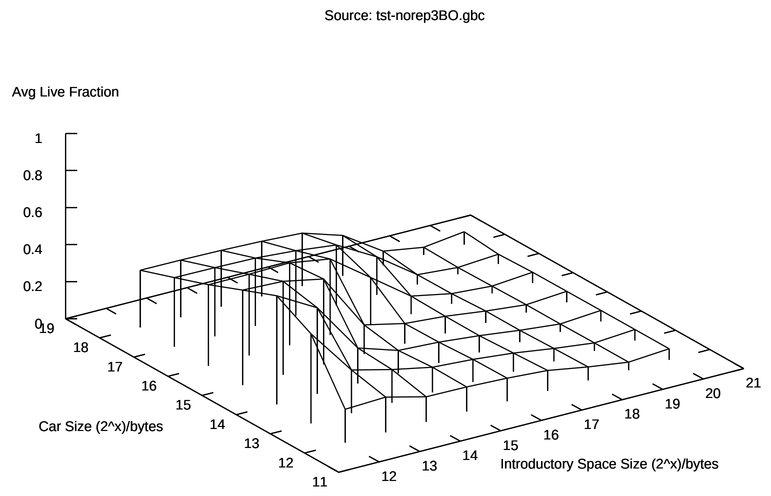


Figure 7.11.: *Average live fraction in the train space executing* `tst-norep3BO.gbc`

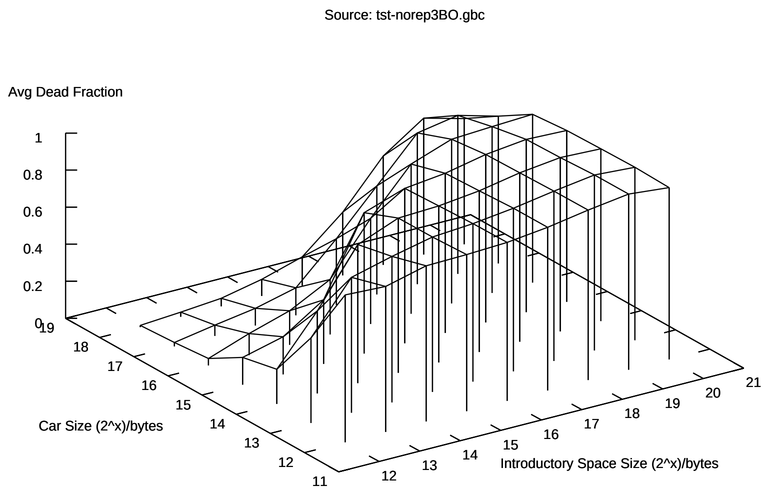


Figure 7.12.: *Average dead fraction in the train space executing* `tst-norep3BO.gbc`

Looking at the live graph for `dfstree.gbc` (see figure 7.14, most of its train heap is live, hardly any is dead, and the unused fraction grows as the size of the car grows. This is not unexpected since most of the bytes allocated in this program are kept live. `constAlloc.gbc` places itself between `dfstree.gbc` and `tst-norep3BO.gbc` with a higher fraction of dead objects than `dfstree.gbc`.

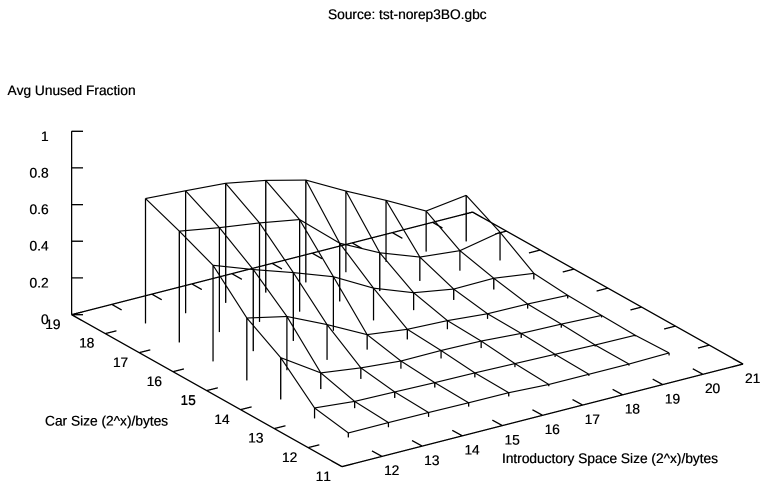


Figure 7.13.: *Average unused fraction in the train space executing* `tst-norep3BO.gbc`

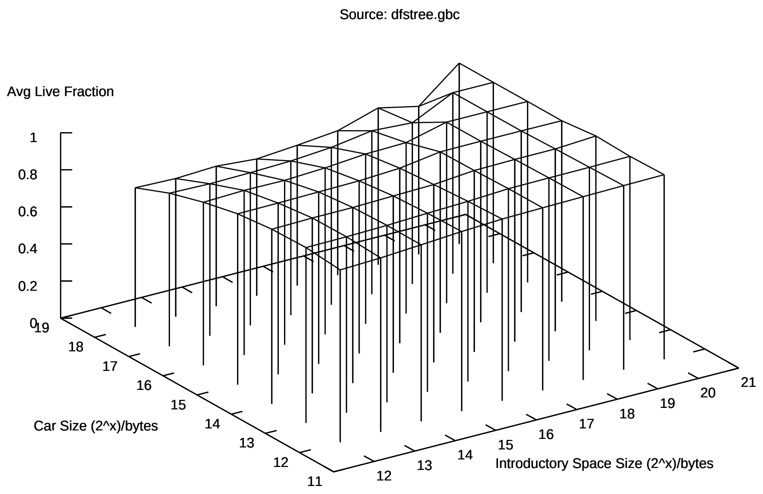


Figure 7.14.: *Average live fraction in the train space executing* `dfstree.gbc`

Including the introductory space with the train generation yields different results. The introductory space is filled with dead and live objects just before garbage collection and unused right after. This gives a time average of 50% unused while the live and dead share the other 50%. The effect of this is that the unused fraction is pulled towards 50% when combined with the train space, especially as the size of the introductory space grows (see figure 7.15). This effect is also visible in most of the test files with the dead fraction. The two test programs with low memory requirements seem to be punished in their live fraction by a larger introductory space; these programs do not require large introductory spaces.

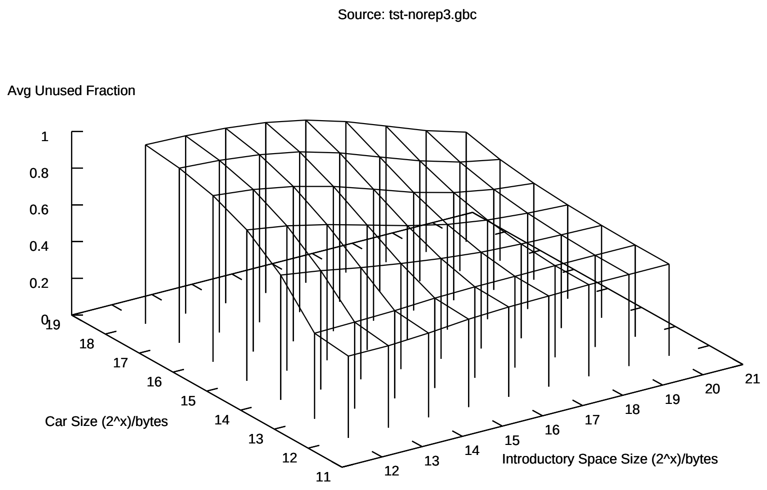


Figure 7.15.: *Unused fraction in the introductory and train spaces executing* `tst-norep3.gbc`

7.5. Time Performance

In this experiment the time-performance impact of variations in the car and introductory space sizes is measured. This includes the average disruption time, the maximum disruption time, the total garbage collection time, and the total garbage collection time fraction of the total execution time.

7.5.1. Method

We try all size combinations in the selected ranges or car and introductory space sizes and measure the max disruption time, mean disruption time, total garbage collection time, and total garbage collection time fraction of the total execution time in each test. Each of the tests is run 3 times with the same car size, introductory space, and test file. The average of the results is used for a single point in one of the graphs.

Executables were compiled with optimization set to `-O2`, a fairly high level of optimization, though not the highest possible.

The five source files `test-norep3.gbc`, `test-norep3BO.gbc`, `allocator.gbc`, `constAlloc.gbc`, and `dfstree.gbc` were all run through this test with both the old and the new train creation policy.

The results are obtained by instrumenting the virtual machine to print the user time used for each garbage collection obtained with the `times()` system call. The total execution time is obtained using the Unix command `time`, subtracting an approximation of the time used for measuring time for the garbage collections (described in 7.5.2). Of the three times reported: real, user, and system; the user time was used for all our measurements.

Finally, the results are plotted into four 3D graphs with the exponents of the introductory space size and the car size as the two independent variables.

7.5.2. Uncertainty

To reduce the uncertainty related to time measurement, each test was run three times, and the average of the results was used.

To reduce the effect of the time report printouts affecting the user time of the Unix `time` command, we approximated this time by running a tight loop that did the same calculations as the time measurement. Dividing this by the number of printouts gave us an approximation of the time cost of one printout. This is important because this reduces the extra measurement cost placed in particular on the test runs with a high scavenging frequency.

7.5.3. Results

The test were run with both the new and the old algorithm, but here we focus on the results of the new algorithm. All graph can be seen in appendix C.

Looking at the average disruption times they range from about zero to about half a second in all tests. Both `tst-norep3.gbc` and `allocator.gbc` have a average disruption time near to zero seconds.

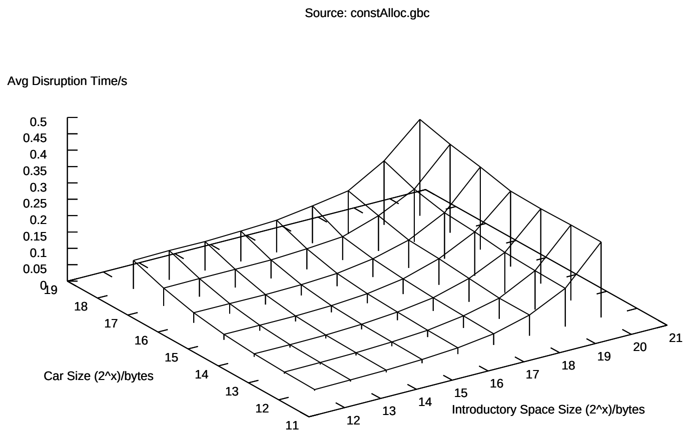


Figure 7.16.: *The average disruption time of* `constAlloc.gbc`

In figure 7.16 the average disruption time of `constAlloc.gbc` is increasing when the car and/or the introductory space size is increased. The same correlation between car/introductory space size and the average disruption time can be seen in the average disruption time graphs of `dfstree.gbc` and `tst-norep-3BO.gbc`, although the trend is not so pronounced with `tst-norep3BO.gbc`.

The maximum disruption time graphs show almost the same characteristics as the average disruption time graphs. Both `tst-norep3.gbc` and `allocator.gbc` have a maximum disruption time below 0.05 seconds.

In figure 7.17 the maximum disruption time of `constAlloc.gbc` is increasing when the car and/or the introductory space size is increased. The same correlation between car/introductory space size can be seen in the maximum disruption time graph of `dfstree.gbc`.

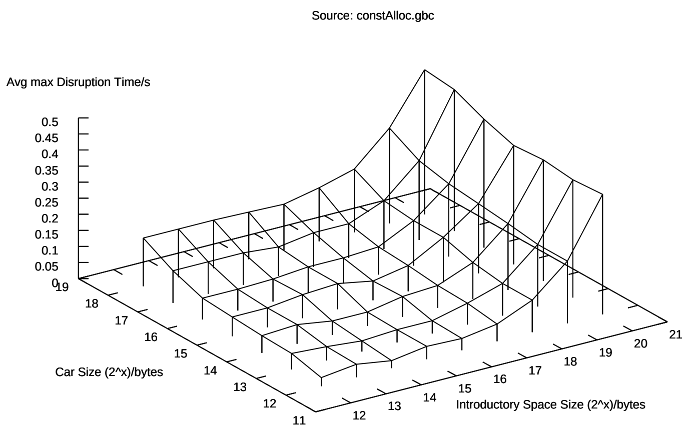


Figure 7.17.: *The maximum disruption time with the source file* `constAlloc.gbc`

The maximum disruption time graph of `tst-norep3BO.gbc` (see figure 7.18) is more indistinct. It seems that the lowest maximum disruption time is present in the introductory space and car size combinations: (4KB, 8KB), (4KB, 16KB), (8KB, 16KB). Another characteristic of the graph of `tst-norep3-BO.gbc` is that the introductory space and car size pairs (65KB, 16KB), (128KB, 32KB), (256KB, 64KB), (512KB, 128KB) have a larger maximum disruption time than their direct neighbors, i.e., the pairs in the graph directly adjacent two these pairs. The mentioned pairs follow a diagonal in the graph. At other diagonals, e.g., the one starting from (4KB, 8KB) and ending with (128KB, 256KB), the garbage collector has lower maximum disruption times than the direct neighbors. We do not have any explanation of this but we suspect that with some introductory space and car size combinations the object structures fits better into the blocks whereas other combinations scatters the object structures over more blocks. When the objects are scattered over more blocks, it will yield lower maximum disruption times since less objects have to be copied at each garbage collection.

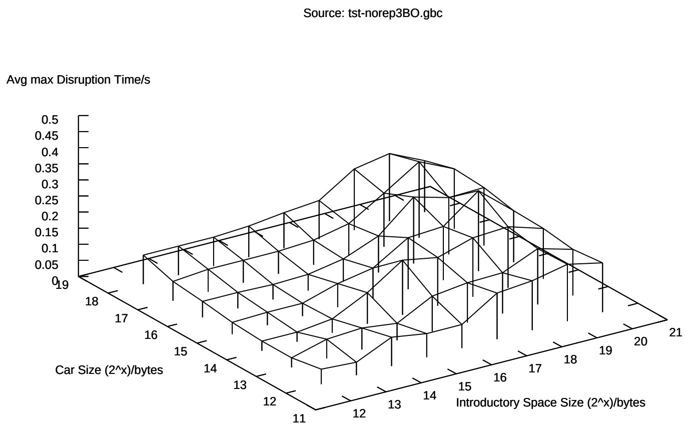


Figure 7.18.: *The maximum disruption time with the source file* `tst-norep3BO.gbc`

The general trend that both maximum and average disruption times are increased with car and/or introductory space size makes perfect sense. An increase in the block sizes implies that the average amount of data that has be processed is increased, i.e., more objects will have to be copied from the introductory space and from car in average.

In general the largest total disruption times is present with a small introductory space size combined with a large car size. This trend is most evident in the graphs of `tst-norep3BO.gbc`, `constAlloc.gbc`, and `dfstree.gbc` (see figure 7.19).

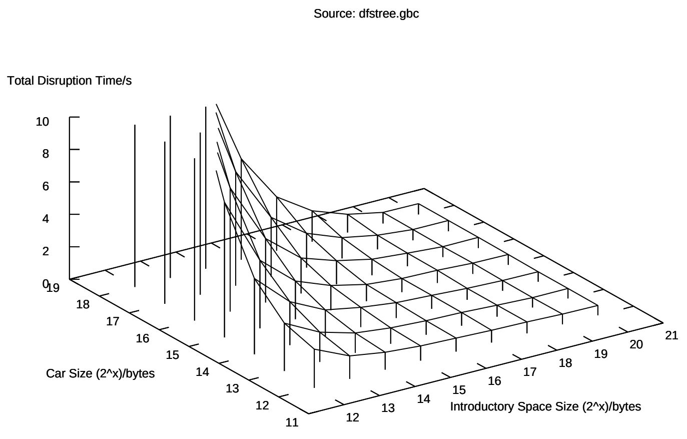


Figure 7.19.: *The total disruption time with the source file* `dfstree.gbc`

This is reasonable since a smaller introductory space size the increases the garbage collection frequency. With a high garbage collection frequency and a large car size the probability of moving a lot of live objects is high. In essence if car size is much bigger than introductory space size, the total garbage collection time will be high.

It should be noted that it seems as if the total disruption time only increases with a decrease in the introductory space size in the graphs of `allocator.gbc` (see 7.20) and `tst-norep3.gbc`, i.e. the total disruption time has little correlation with the car size in these test program executions. This trend is not that pronounced though, but the reason for it most probably is that these test programs keep very few bytes alive. This implies that very little object copying has to be done during these test program executions.

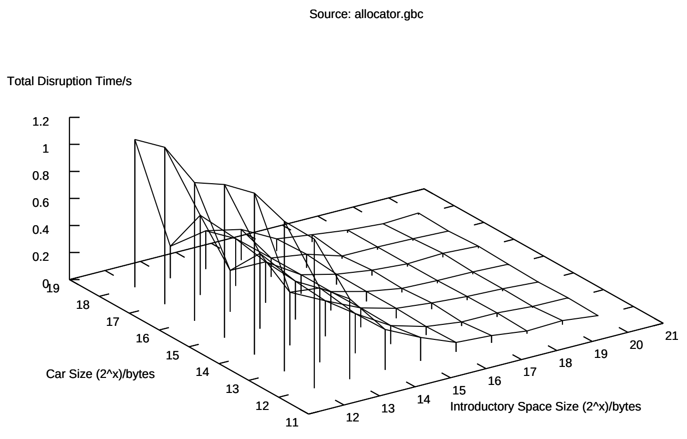


Figure 7.20.: *The total disruption time with the source file* `allocator.gbc`

The garbage collection time fraction graphs of `tst-norep3.gbc` and `allocator.gbc` confirms that little live object copying has to be done when gbvm executes these test programs, and again, the garbage collection time fraction is independent of the car size.

When gbvm executes the other test programs, the total garbage collection fraction is increased with an increased size of the car and introductory space size. This is most pronounced in the execution of `dfs-`

`tree.gbc` (see figure 7.21) but the same tendency is evident in the execution of `tst-norep3BO.gbc` and `constAlloc.gbc`.

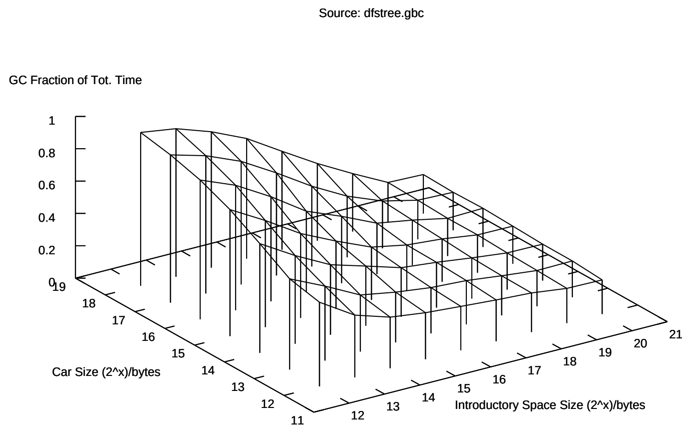


Figure 7.21.: *The garbage collection time fraction of the total execution time with the source file* `dfstree.gbc`

Looking at the graphs it is obvious that something happens in many cases when crossing the diagonal where the sizes of the introductory space and cars are equal. If the test program keeps many objects alive, both the total disruption time and the garbage collection time fraction are high in the left triangle (where the car size is equal to or larger than introductory space size) and low in the right triangle (where the car size is equal to or smaller than introductory space size). This can be explained with the following postulate: When one decreases the garbage collection frequency by doubling the introductory space size, one can double the car size too and still end up with the same garbage collection time fraction. The reason is that approximately the same amount of heap space is scavenged in total; the scavenged heap size is doubled, but so is the expected time before the next scavenge. In the figure this explains the almost equal height of the lines parallel to the mentioned diagonal. That these lines are generally higher towards smaller sizes is reasonable since using smaller sizes will typically induce extra write barrier overhead and there is probably also a base cost of conducting a garbage collection no matter what size. One should remember, though, that traveling along these lines affects the disruption times.

7.6. New Train Policy

The creation of a new train is considered in our system when a non-moved object referenced from the stack space is found, but creating a new train for each such object would lead to too many new trains for the garbage collector to keep up with.

With our initial method one new train is created each garbage collection if such an object is found. Unmoved objects found later are moved to the last car of the last train. We initially believed this was a good way to introduce new trains as these could later grow larger if they referenced other objects. As this can sometimes lead to cars that are not filled sufficiently, this method induces higher memory requirements than necessary.

This problem leads us to restrict the policy for creating a new train. The constraint of only creating one new train per scavenge was kept, but this is only done if either the last train contains more than one car or the first car in the last train would be filled more than a fixed fill threshold if an object is added. In this section we will investigate the significance of this change to the train algorithm implementation.

7.6.1. Method

The space and time tests discussed in sections 7.4 and 7.5 are all performed with the new and the old train creation policy. This gives us a large set of data to evaluate the effect of the train creation policies both with regard to space and time consumption. The methods for performing the tests are identical to those previously described.

The fill threshold has been set to 80% in all the experiments with the new train creation policy. It would also be interesting to optimize this threshold in further experiments, but this has not been done since we had to stop somewhere, and this would require a lot of work and space on our test hardware.

7.6.2. Uncertainty

We are not aware of any uncertainties introduced in addition to those mentioned in sections 7.4 and 7.5.

7.6.3. Results

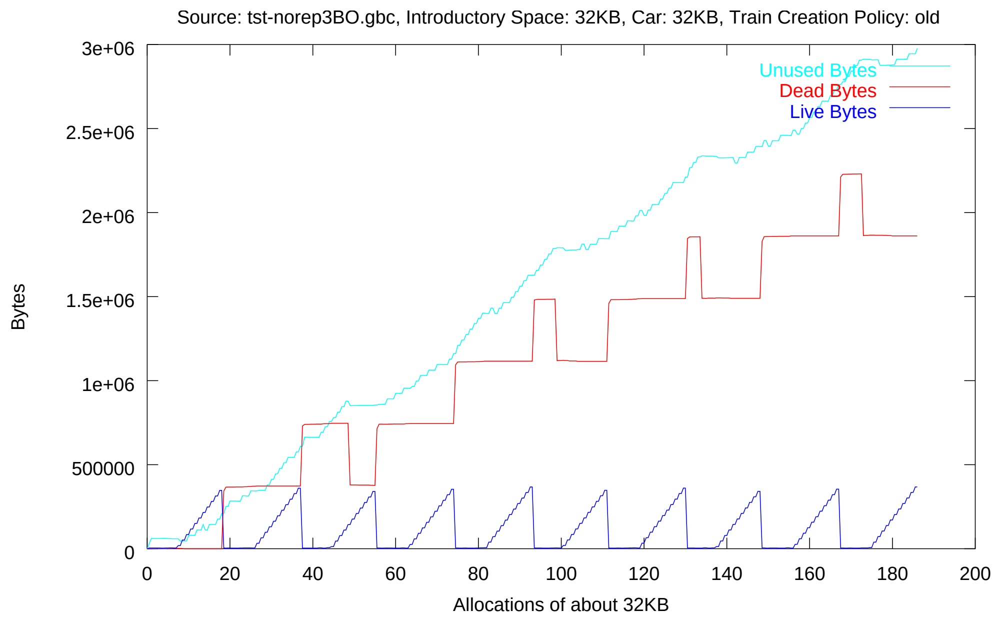


Figure 7.22.: *Allocation with the old method, introductory space and car sizes 32KB.*

The new train creation policy has a profound effect on some of the tests while others are virtually unchanged. An example of a significant (but not the greatest) effect can be seen in figures 7.22 and 7.8.

In this experiment a large chunk of data is allocated ten times and the garbage collector is trying to keep up. Particularly the unused space is improved a lot in the new version, but also the amount of space consumed by dead objects is reduced notably. An interesting thing about this particular test file, `tstnorep3BO.gbc`, is that it allocates a lot of objects that are kept live during the "big object test". These are suddenly all released and a series of garbage collections with a low number of live bytes follows. It would seem from the graphs that these chunks are always collected together. This makes it easy to see how many large garbage structures the garbage collector has removed; the old one reclaims four out of nine while the new one reclaims six of nine, two of which are reclaimed in one scavenge.

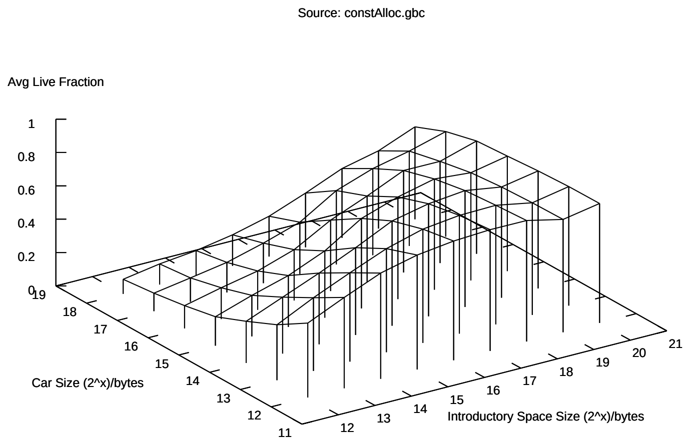


Figure 7.23.: *The fraction of live data in the train space with the* `constAlloc.gbc` *using the old train creation policy.*

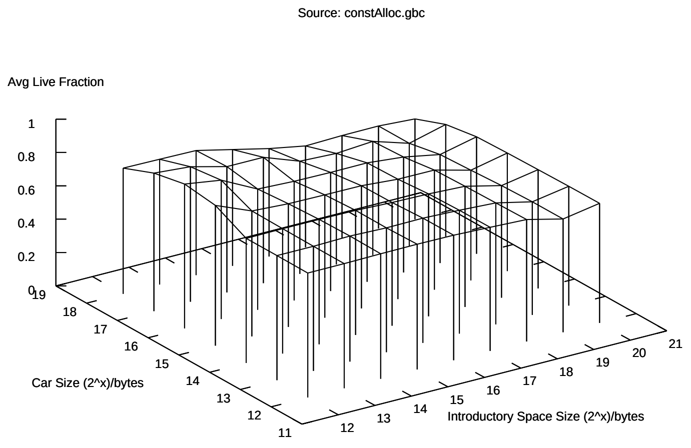


Figure 7.24.: *The fraction of live data in the train space with the* `constAlloc.gbc` *using the new train creation policy.*

The graphs 7.23 and 7.24 show one of the examples of significant differences between the amount of live objects in the train space caused by the new train creation policy. It is interesting to note that the region where the old version has the worst space performance (with cars larger than the introductory space) the new version has its best space performance. The reason for the behaviour of the old version is a lot of unused space in newly created cars; because the introductory space is smaller than the car size, new cars are never filled. The new version is particularly good at removing dead objects in this region because it does not scatter its objects over as many cars and because the garbage collection frequency is high there.

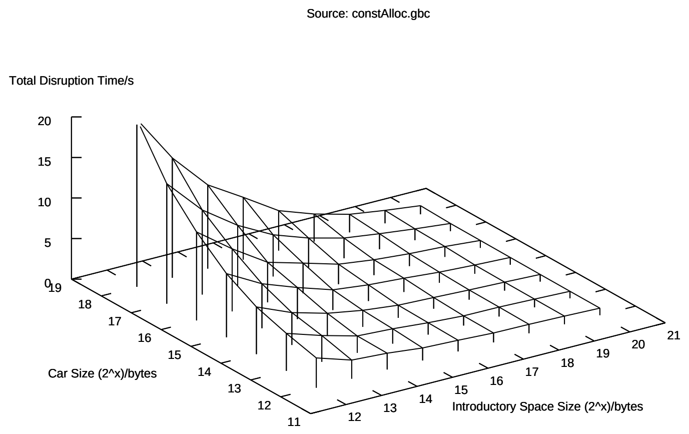


Figure 7.25.: *The total garbage collection time with `constAlloc.gbc` using the old train creation policy.*

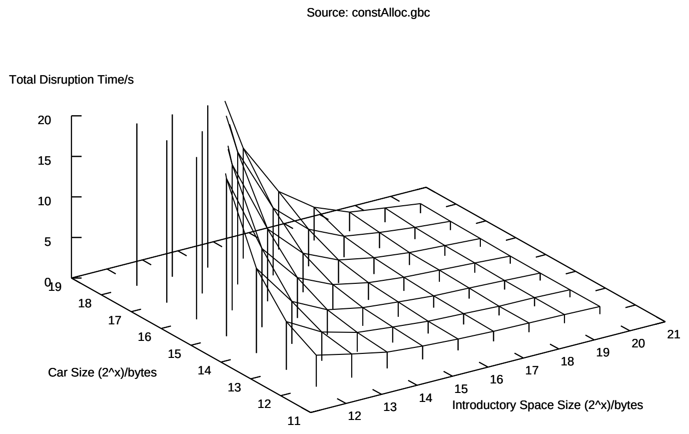


Figure 7.26.: *The total garbage collection time with `constAlloc.gbc` using the new train creation policy.*

Primarily the experiments with lots of live objects are influenced by the time overhead caused by the new algorithm variation see figures 7.25 and 7.26. The change is significant in the region where the car size is larger than the introductory space size.

Where the new train policy saves space, it seems to costs in time. We believe the reason for this is that more live objects are present in each car and these have to be moved during garbage collection. However, one could argue that the old train creation policy does not do the same amount of work as the new one since it is slower at reclaiming the dead objects. To do the same work, it requires more scavenge iterations, adding an (undetermined) time penalty to the old train creation policy. Having the same number of objects in a heap divided into more segments is not an advantage with regard to quickly reclaiming garbage structures. For these reasons we believe the new train policy is an improvement over the old policy.

7.7. Speed Comparison

Although we have used a lot of energy to hide this fact, the real purpose of this project was to make the fastest (garbage collecting) gbeta virtual machine. This experiment will reveal if this purpose has been fulfilled.

7.7.1. Method

Using the test programs listed in 7.1, we compare the time used by the three gbeta executing systems known to exist: gbeta-0.81 [Ern99], gbetai [JJW01], and our own system (gbvm). To make the results more reliable we ran each source file a hundred times and calculated the mean time used. gbvm was compiled with full optimization, an introductory space size of 64KB, and a car size of 32KB. The choices of car and introductory space sizes were made as they are as they are the middle values in our ranges, and they seem like reasonable compromises too.

To measure how much time gbvm used we used the Unix command `time`. As shown in the previous experiments the performance of gbvm varies with the selected introductory space size and car size. So, if we had selected different sizes gbvm could have performed differently.

gbeta-0.81 has a special option (`-r`) to make it generate code eagerly and do run time measurements. If we used the output from the `time` command instead of `-r`, the time used to compile the gbeta source file would be included in the running time. In addition to this, the option `-l` was added which makes gbeta perform static analysis lazily.

To measure the time used by gbetai, we also had to exclude pre-compiling time. In gbetai they compiled the gbeta byte code into Java source files. These source are then compile using a standard Java compiler (`javac`), and the resulting Java byte code can be run using a Java virtual machine. So, to be fair we measured the time used by a Java virtual machine to execute the Java byte code resulting from gbeta byte code compilation as this is the time a user would experience.

Our usual test-programs were not supported by gbetai, so we had to make a special program (`simple.gbc`) to compare gbvm with gbetai. Besides this program we used a number of benchmark programs used in gbetai [JJW01] to compare their virtual machine with BETA. These benchmarks are small gbeta programs that test one specific thing repeatedly such as assignment, specific steps in run-time path traversals, and usage of virtual attributes:

`assignment`	Assignment of value to integer object.
`objRefAndIns`	Assignment of one object reference to another and an instantiation of an object.
`object`	Instantiation of an object.
`runtimePath0`	Traversal of empty runtime path.
`runtimePath1`	Traversal of run-time path with one lookup step.
`runtimePath2`	Traversal of run-time path with one indirect lookup step.
`runtimePath3`	Traversal of run-time path with one up step and one lookup step.
`runtimePath4`	Traversal of run-time path with two up steps and one lookup step.
`runtimePath5`	Traversal of run-time path with one out step.
`virtual`	Invocation of a virtual attribute.

7.7.2. Uncertainty

We are unaware of any uncertainties in addition to those of the other speed measurement experiments.

7.7.3. Results

	gbvm/s	gbeta-0.81/s	Speedindex	gbetai/s	Speedindex
tst-norep3	4.777	39.020	8.169	-	-
cruncher3	3.749	33.031	8.810	-	-
allocator	7.766	109.639	14.118	-	-
constAlloc	6.218	64.758	10.415	-	-
dfstree	3.153	27.652	8.772	-	-
derive	5.089	194.840	38.287	-	-
simple	0.063	-(*)	-	3.180	50.232
assigment	6.480	95.356	14.715	3.628	0.560
objRefAndIns	1.897	23.716	12.502	2.824	1.489
object	3.784	106.460	28.133	3.172	0.838
runtimePath0	4.785	85.530	17.874	3.250	0.679
runtimePath1	6.402	102.728	16.046	3.332	0.520
runtimePath2	6.522	102.294	15.684	3.635	0.557
runtimePath3	7.509	115.910	15.436	3.761	0.501
runtimePath4	8.020	117.244	14.619	4.168	0.520
runtimePath5	6.807	113.276	16.642	1.468	0.216
virtual	50.033	419.866	8.392	42.394	0.847

Table 7.1.: *Speedindex is the time used by gbeta or gbetai divided by time used by gbvm. So a speedindex of 0.5 means twice as fast as gbvm while speedindex 2 means half as fast as gbvm. (*) gbeta failed to execute this program successfully on the test machine.*

Looking at the results it seems that our gbvm are competitive with the other gbeta executing systems. With gbvm having index one the index of gbeta-0.81 varies from 8.169 to 38.287. This were expected since the main purpose of gbeta-0.81 is to prove that the gbeta language can be implemented not that it can execute fast. One must also take into account that gbeta-0.81 supports the compilation and execution of any given gbeta program, whereas we do only support a subset of all gbeta programs. For instance, our `Thread` class should be modified to fully support concurrency and thereby synchronization which could incur some run-time overhead.

The comparison between gbvm and gbetai is interesting. With `simple` and `objRefAndIns` gbvm are faster than gbetai, but in all other cases gbvm is slower ranging from speedindex 0.216 to 0.847. Three things can be concluded from these results. Firstly, gbvm is competitive overall even though the virtual machine component has not been optimized. Secondly, the experiment shows that it is possible to optimize a gbeta virtual machine. Without knowing the details of gbetai, it seems that gbetai is fastest in the benchmarks where an optimization effort was made. Thirdly, we did not succeed making the fastest gbeta virtual machine.

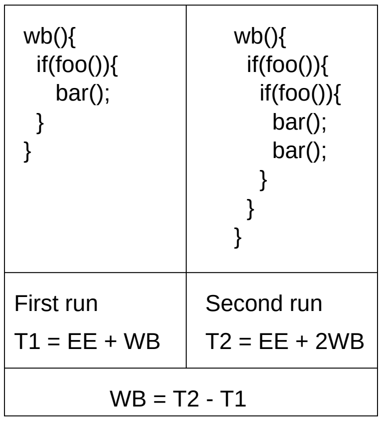
7.8. Write Barrier

The purpose of this experiment is quantify the amount of time used in our write barrier.

7.8.1. Method

One way to quantify the time used in the write barrier is profiling. `gprof` [GKM82] could be used, but it will not work if inlining is used. Since we do want to inline the write barrier code, we cannot use `gprof` to quantify the time used in the write barrier.

Another way to do the write barrier time quantification is to compare two runs of the same program. The first run with a normal write barrier and the second with a write barrier that does exactly the same, semantically, but executes each instruction twice (see figure 7.27).



```
wb(){
  if(foo()){
    bar();
  }
}
```
	```
	wb(){
	  if(foo()){
	    if(foo()){
	      bar();
	      bar();
	    }
	  }
	}
	```

First run T1 = EE + WB	Second run T2 = EE + 2WB
WB = T2 - T1	

Figure 7.27.: *One way to quantify the write barrier.* T1 *= time used by the first run,* T2 *= time used by the second run,* WB *= time used by write barrier,* EE *= time used by everything else*

To quantify the write barrier we wrote a version of the write barrier macro that did everything twice without changing the effect of the write barrier. We then measured the user times of an executable running with a single write barrier executing the listed source files (see table 7.2), and the user times of an executable using the double write barrier macro. The estimate of the time consumed by the write barrier is the difference between these execution times. The executables were compiled to have an introductory space size of 64KB and a car size of 32KB. Again, these sizes were chosen since they are the middle values of our tested introductory space size and car size ranges.

7.8.2. Uncertainty

As with the speed comparison there is an uncertainty connected with the fixed car and introductory space size. It may very well give another result if different car and introductory space sizes had been chosen.

Another uncertainty is the characteristics of the source programs. Some of them like e.g., `derive.gbc` do not use the write barrier much whereas others use it heavily. This is only a problem if the write barrier light programs run for a short period of time inferring a great uncertainty in the measured user time. The difference, in user time, of the two executions could then be caused by normal time measurement uncertainties instead of the double write barrier.

The hash set used for the remembered sets has a resize functionality that may and may not be invoked during the tests. If a resize happens in the single write barrier case it will also happen in the double write barrier case but only once in each cases. This means that the cost of resizing the hash map is not included in the measured time difference. If the resizing was instead performed twice in the double write barrier case, the results of this experiments would have been more accurate, but this is difficult to do without altering the implementation in ways that affect the performance in other ways.

Another problem with the remembered set hash sets is that the second time the same value is inserted, no write operation is performed. This is an advantage for the double write barrier macro and it may therefore be faster in its second execution.

Caching effects will also increase the speed of the second execution of the write barrier.

To sum up it seems that we should expect to get too small time measurements by using this method.

7.8.3. Results

	Single write barrier/s	Double write barrier/s	Write barrier time
tst-norep3	4.765	5.013	5.198%
cruncher3	3.700	4.046	9.357%
allocator	7.752	8.536	10.115%
constAlloc	6.144	6.621	7.760%
dfstree	3.110	3.397	9.205%
derive	5.090	5.443	6.940%
simple	0.062	0.074	19.548%
Mean:	-	-	9.732%

Table 7.2.: *Results of running the the different source files with and without a double write barrier. Write barrier time is $(T2-T1)/T1$ in percents*

Our write barrier seems to be acceptable overall. Between 5% and 20% of the time is used in our write barrier, with a mean of about 9%. The rather diverting result of `simple.gbc` could be due to the before mentioned uncertainty related to the short execution time.

The validity of this experiment is a bit questionable though. The quantified write barrier time should have had the total time used by resizing added. This is only a problem if a lot of remembered set resize operations is present.

The virtual machine component has not been optimized and after having parsed the input file it interprets instructions. This means that the small percentage of time used in the write barrier could be due to a slow virtual machine component. Profiling the system will reveal if this is the case. We present the results of profiling gbvm in the next section.

7.9. Profiling

The purpose of this experiment is to measure the fraction of time used by the two components in our system, namely the virtual machine component and the memory management component. This will set the write barrier experiment into perspective since the low percentage of time used in the write barrier could be due to a slow virtual machine component.

7.9.1. Method

Using the in table 7.3 listed source files we used instrumented our executables and used `gprof`. gbvm were executed a hundred number of times with `gprof` and the result were summed up by `gprof`.

7.9.2. Uncertainty

The results of `gprof` are subject to statistical uncertainty but this has been decreased by making `gprof` sum up the results of hundred executions of each source file.

7.9.3. Results

	ByteCodeLoader.parse()	Thread.run()	GarbageCollector.gc()
tst-norep3	7.6%	69.0%	1.3%
tst-norep3BO	3.3%	53.8%	30.5%
cruncher3	0.2%	52.4%	38.2%
allocator	0.0%	81.6%	0.7%
constAlloc	0.2%	56.9%	31.7%
dfstree	0.4%	53.7%	36.0%
derive	0.0%	77.5%	gc() not invoked
simple	0.6%	73.1%	gc() not invoked

Table 7.3.: The parsing, instruction execution and garbage collection time fractions as `gprof` reports it after 100 runs of each source file

In table 7.3 we list the time fractions used by: parsing (ByteCodeLoader.parse()), executing instructions (Thread.run()), and garbage collection GarbageCollector.gc(). If one adds up the three numbers one does not get 100%, since the execution time is also used for other things such as initialization, but these take only a small fraction of the total execution time.

Looking overall at the profiling results it seems that our virtual machine component takes a large fraction of the total execution time, and a general optimization could increase the write barrier fraction of total execution time. To optimize this component it is necessary to look at the time used in different parts of the virtual machine. Looking further at the output from `gprof` we identified a number of time consuming operations. Among these were:

`Dynamic Casts` Used amongst other things after run-time path traversals, yielding more secure but less time efficient code.

`Run-time Path Traversals` Especially up and down steps are expensive, since they require linear search in the number of part objects in the current object.

`Pattern Instantiation` When a pattern is instantiated all its mixin must be instantiated into part objects put into an object.

`MainPart Lookup` As noted in the evaluation of the virtual machine (see section 5.7) when a specific main-part is wanted all the main-parts are searched linearly.

If a more efficient virtual machine is wanted these operations should be considered for optimization.

7.10. GBVM vs. JVM

The purpose of this experiment is to compare the efficiency of gbvm with Sun's Java HotSpotVirtual Machine (jvm) version 1.3.1.

7.10.1. Method

To compare the two virtual machines we rewrote `cruncher3.gb` to a less gbeta exploiting version (`cruncher3JAVA.gb` - see appendix A). We then tried to write a semantically equivalent program in Java (`cruncher3.java` - see appendix A). Finally we measured the time used by gbvm to execute `cruncher3JAVA.gb` and compared it to the time used by jvm to execute `cruncher3.java`.

To make a representative result we ran each subexperiment a hundred times and calculated the mean value. gbvm were compiled to have an introductory space of 64 KB and a car size of 32 KB.

7.10.2. Uncertainty

We are aware that it is difficult to compare different programming languages and do them all justice. One of the problems in this experiment is that gbeta has a more fine-grained way of storing objects, i.e., in gbeta an object is divided into one or more part objects which must be searched during run-time path traversals. The more fine-grained object representation yields both more run-time flexibility but it also a run-time overhead.

Another problem is that we have not in any way proven that the two programs are semantically equivalent.

The overhead of the jvm must be taken into account. Although gbeta is perhaps richer than Java the support for, amongst other things, concurrency and synchronization in jvm must be a disadvantage compared to our single threaded virtual machine.

Another discrepancy is the type of the virtual machines. Our virtual machine is an interpreter whereas the jvm is a dynamic compiler.

7.10.3. Results

	Time/s	Speedindex
java cruncher3	1.831s	4.732
gbvm cruncher3JAVA.gbc	0.387s	1.000

Table 7.4.: Comparison between jvm and gbvm

As shown in table 7.4, jvm is almost five times slower than gbvm in the given case. This may be because of the startup time of jvm and also synchronization and concurrency support could incur a run-time overhead in jvm.

To give jvm the opportunity to benefit from dynamic compilation we made a new longer running version of both `cruncher3.java` and `cruncher3JAVA.gbc`. The results of this new experiment are presented in table 7.5.

	Time/s	Speedindex
java cruncher3Long	1.8086s	0.546
gbvm cruncher3LongJAVA.gbc	3.3029s	1.000

Table 7.5.: New comparison between jvm and gbvm, with more time demanding versions of the test programs.

The new comparison gives a rather different result. Now jvm suddenly outperforms gbvm. Since jvm is a dynamic compiler it benefits from compiling often invoked methods. The characteristics of the test program fits perfect to dynamic compilation, since the same methods are executed numerous times.

Another interesting detail is that jvm uses less time on a version of the same program that executes more iterations. This could be because its profiler decides to optimize more when the number of iterations in loops are increased which is the case here.

7.11. Discussion of the Results

In this section we will summarize and discuss the interesting results from the previous sections with the purpose of trying to find a good general purpose compromise between the settings tried, as well as suggesting possible improvements.

To comment the speed comparisons with the other systems first, gbvm is faster than gbeta and competitive with gbetai which is about twice as fast as gbvm. The write barrier tests show an average consumption of less than 10% of the total execution time. These results are quite satisfying to us.

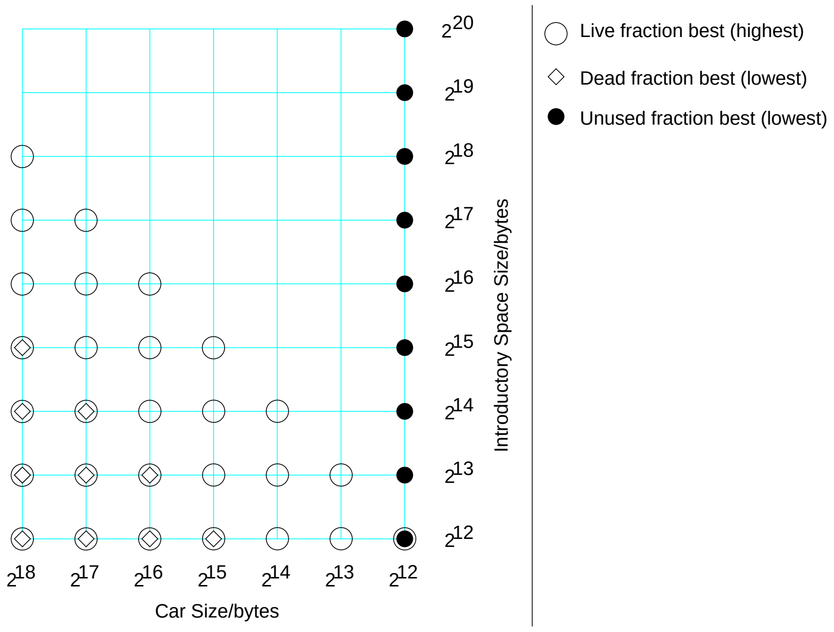


Figure 7.28.: *Summary of the results from the space utilization experiments*

In figure 7.28 we have summarized the results from the space utilization experiments when focusing on the train space. The live fraction is highest when the introductory space is smaller or equal to the car size. The dead fraction is lowest when this heuristics is tightened so the car size must be at least 8 times larger than the introductory space. Finally, the unused fraction is best with the smallest car size. In addition to this the lowest of the car size and the introductory space size limits the size of objects that can be handled in the system.

Figure 7.29 summarizes the results from the time evaluation experiments. The average and maximum disruption times are lowest as long as the combined size of the scavenged space is not too large while the total disruption time is lowest when the introductory space is larger than the car size.

With regard to train creation policy, the new train creation policy seems to be the best overall of the two

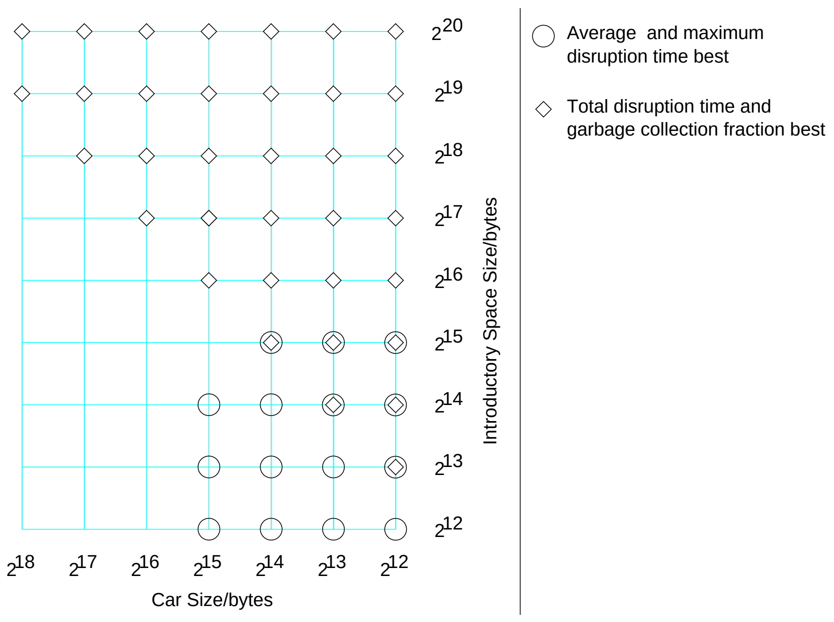


Figure 7.29.: *Summary of the results from the time performance experiments*

proposed possibilities since it does not waste as much space as the old train creation policy, and it is questionable whether the old policy is faster anywhere if both versions have to reclaim the same amount of garbage.

Combining the space utilization and time performance experiments for a good compromise is not easy. If one has to make such a choice we would recommend approximately equal introductory space size and car size with a value that allows a reasonable size of objects e.g., 32KB. It is possible to set the sizes so most large garbage structures are reclaimed quickly, but then one will pay for it with high garbage collection overhead and wasted space. It will probably always be possible to put so large structures into the system that performance will suffer. It is also possible to get low overhead from garbage collection by keeping the car size small and adjusting the disruption time with the introductory space size, but still keeping it reasonably large. Such a system will not be good at reclaiming large garbage structures at a reasonable pace, but it will *eventually* reclaim each structure (if a zero progress situation [GS93] or out of memory fault occur).

For the best performance it is necessary to adjust the virtual machine parameters to values that fits the application. This is the best way in our framework to avoid doing *defensive garbage collection* that is, doing more garbage collection than seemingly necessary in order to reclaim large garbage structures rapidly when they are created.

The biggest problem as we see it in the context of finding a good compromise of the parameters is that it is not generally possible, as we believed, to combine a large introductory space with a smaller car size without risking inefficient memory usage. A large introductory space is supposed to filter out many short lived objects before they even reach the train space, but it also causes a low scavenging frequency which makes the system less responsive. When the assumption of many short lived objects does not hold, our collector in this configuration easily gets in trouble when large chunks of objects die. The total garbage collection time fraction is still low in the large introductory space and small car size areas of the graphs, so it may still be possible to construct a better performing garbage collector if a different strategy was chosen for selecting not only the scavenging *frequency* but also the scavenging *moment*.

Examples of unexplored (by us at least) strategies for a better heuristic of the moment for garbage collection could include the number of pointer updates which are not initializing stores since this could be an indication of possibly new dead objects. This would probably cost in the write barrier, but finding very good times for doing garbage collection might even this out, and it may be possible to find a cheap approximation such as a simple count of pointer stores outside the introductory space. Another estimate for the number of dead bytes in the system could be inferred from the number of bytes rescued last time. With such dead byte indicators, alone or combined, one could try to control the rate of scavenging using this number in an adaptive scheme.

In this adaptive scheme the memory manager needs to be able to adjust the amount of scavenged data in order to control the responsiveness of the garbage collection in the train space. Continuing with our current setup, one can vary the number of cars being scavenged together with the introductory space from zero to many. This would approximate the effect of being able to change the size of cars and thus enable us to move in one dimension in the 3D graphs. Moving in the other dimension can be attained by allocating a rather large introductory space and setting a dynamic fill limit.

One could consider if a filled up introductory space really means that one can expect to find a lot of dead objects in the system? Perhaps in the introductory space, but in the train space this is not necessarily the case. While a large garbage structure is being built, the garbage collector is busy trying to reclaim objects, but after the intensive allocation ceases, the effort of the garbage collector ceases too. Why not try to do the garbage collection after the intensive allocation has evened out? A simple timer could check periodically if a garbage collection has occurred during the last period and trigger a garbage collection if a number of instructions have been executed or some other sign of activity in the system is showing.

One could also ask if a high fraction of live objects really is a good sign?. It could indicate that too much emphasis has been put on garbage collection perhaps doing too defensive garbage collection. We see that the total amount of garbage collection time increases a lot if we try hard to keep the train space clean of dead objects. One could also take a more relaxed attitude to the memory consumption, hope that programs will not allocate memory like crazy just to forget about it a moment later and start all over (like some of our test programs). After all there is a concept called programming guidelines which could dictate that this is bad programming style and should be avoided.

8. Related Work

This chapter will give a short overview of research related to this thesis. We compare four memory managers all using a train algorithm. The two topics compared are: Memory layout and pointer tracking scheme. Figure 8.1 gives an overview of the four memory managers in focus.

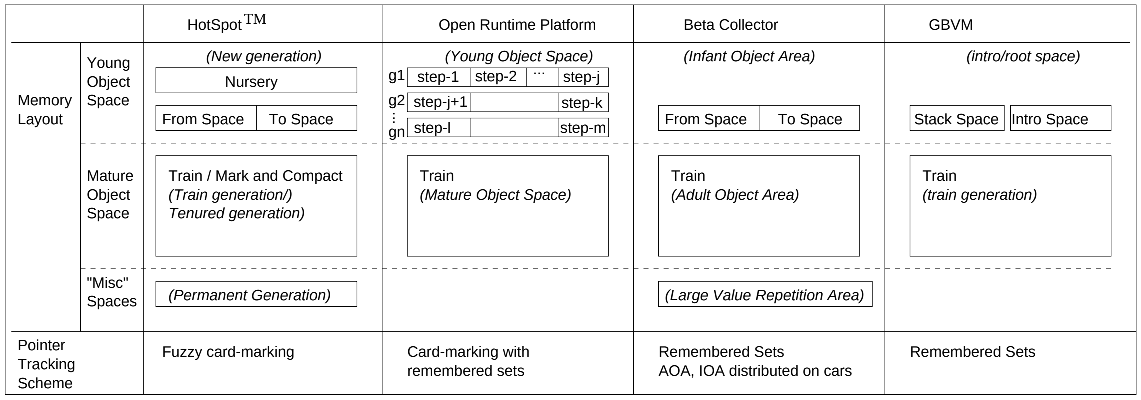


Figure 8.1.: *Overview of memory layout and pointer tracking scheme. Text in italic describes the names used in the different memory managers*

8.1. HotSpot

HotSpotis the well known Java virtual machine developed by Sun Microsystems, Inc.. Some of the people behind the memory management in HotSpotare the people behind the incremental train garbage collector for BETA [GS93]. The principles behind HotSpotis further described in subsection 3.3.

8.1.1. Memory Layout

As figure 8.1 illustrates HotSpothas two subspaces in Young Object Space (*New Generation*) according to Lars Bak. A nursery used to allocate new objects and a copy collector space with a 'to' and 'from' space. Objects are allocated into the nursery space and promoted to the copy-collector space. If they survive long enough they get promoted to the Train / Tenured generation depending on how HotSpothas been configured. HotSpotcan be configured to use a mark-compact collector or a train algorithm collector in mature object space (MOS). HotSpothas a Permanent Generation where reflective data (classes, methods, symbols, etc.) are stored. [SM01].

8.1.2. Pointer Tracking Scheme

According to Lars Bak they use a software write barrier with fuzzy card marking. This scheme corresponds to the scheme described in [Höl93].

8.2. Intel's Open Runtime Platform

Intel's Open Runtime Platform (ORP) is an open source platform for experimenting with dynamic compilation and garbage collection technologies [Hud00]. It contains three separate modules: Virtual Machine (VM), Just-In-Time compiler (JIT), and Garbage Collector (GC). The main advantage of the separation between JIT/VM and GC/VM is the possibility to experiment with e.g., garbage collection without learning the entire system. [Hud00]

8.2.1. Memory Layout

Conceptually they divide the memory into three areas: collected, traced, and untraced. The collected area contains all objects (live or dead) which have been allocated by the garbage collector. The traced area may contain objects with pointers referring to the collected area this area may include the run-time stacks, statically allocated data and hardware registers. The untraced area includes data which is ignored by memory management.

The collected area, also called the heap is divided into two spaces. "Young Object Space" (YOS) holding recently allocated objects and "Mature Object Space" (MOS) containing objects that have survived a number of scavenges. They use a generational copy collector in YOS and a train algorithm garbage collector in MOS. Steps are used in the generations to separate differently aged objects.

8.2.2. Pointer Tracking Scheme

In ORP they use a combination of card-marking and remembered sets. They use card-marking to record interesting memory areas (cards) at run-time and summarize the interesting pointers into remembered set at garbage collection time. This scheme was invented by [HH93], who noted that a combination of remembered sets and card marking is more efficient than either remembered sets or cards alone.

ORP uses a card marking algorithm corresponding to the one described in [Höl93]. The following code briefly explains the algorithm:

```
card_table_base[(object_ref->heap_base)>>bits_to_shift]:=MARK;
```

Figure 8.2.: *Write barrier in ORP*

In the card marking table they mark the entry corresponding to the card where the object having the pointer being updated starts (see figure 8.2).

8.3. The Incremental BETA collector

Connected with their master's thesis work at Aarhus University, Steffen Grarup and Jacob Seligman were among the first to implement a train algorithm garbage collector. They implemented an incremental garbage collector for the mature object space in BETA [GS93]. Their main inspiration came from [HM92].

8.3.1. Memory Layout

As illustrated they have an Infant Object Area (IOA) used for allocating new objects. This space is copy collected. Objects surviving a configurable number of times in IOA are promoted to adult object area (AOA). In AOA the train algorithm collector is used. Finally they have a Large Value Repetition Area (LVRA) where non-pointer arrays of integer, char, etc. above a configurable size are allocated and resident.

8.3.2. Pointer Tracking Scheme

Each car/area has a remembered set. A specialty with this implementation is that the remembered set of IOA is distributed out to the cars in AOA. This makes it easy to update the remembered set of IOA when a pointer in a specific car is altered, but it must have a serious drawback, when one has to gather the entire remembered set of IOA. This is only done once each garbage collection of IOA.

The software write barrier uses from 11 to 18 SPARC-instructions for stores in AOA and less in IOA.

8.4. GBVM

Our own gbeta virtual machine with a simple heap layout train algorithm garbage collector, does not need much presentation, since this thesis mainly deals with it.

8.4.1. Pointer Tracking Scheme

We use pr. space remembered sets to record interesting pointers. We have a software write barrier. The algorithm is described in section 6.9.

9. Future Work

If we added a large object space (LOS) to our memory management component it might increase performance. The LOS could store objects with no references (e.g., repetitions) and of course large objects. This would add more freedom to the choice of introductory space size and car size, since objects larger than the smallest of the two could be allocated in LOS.

An adaptive scavenging scheme as discussed in section 7.11 would possibly make the garbage collector better at adjusting its effort to the current demands of the application.

Compared to other related systems, gbvm is the only system which does not have a copy collector in the young object space (our introductory space). It would be interesting to implement a copy collector in introductory space to see how it would perform compared the current setup.

Another working area is extending the virtual machine to make it fully working, i.e., able to execute any given gbeta program. The most important lacks of the current virtual machine are the missing repetitions and concurrency. To make our system multi-threaded a number of things must be changed. Amongst the things are a signaling queue, synchronization of the write barrier, and it would also be necessary to garbage collect the stack space in some way since threads can die before the whole program terminates.

Optimization of the virtual machine component has almost been non-present in this implementation, so there is hope for performance improvements. For instance if the `ByteCodeLoader` made a two pass parse of the input file, we would be able to do a number of optimizations. These would include: caching main-parts in the `AddMainPart` instructions, use a real byte code format.

After having optimized the virtual machine component it would be interesting to compare it with BETA to see how much the added generality costs in time performance.

10. Conclusion

In this project a working gbeta virtual machine (gbvm) has been implemented from the ground up in `C++` with support for a major part of the gbeta byte code instructions. In this implementation automatic memory management has been our primary focus. For this purpose the train algorithm has been deployed in a previously unseen heap layout accompanied only by an introductory space.

The virtual machine component has been redesigned to allow garbage collection safe points even while initializing attributes and executing multi-line instructions.

The memory management component has been almost completely redesigned and reimplemented with more efficient data organization and structures. A powerful memory manager debugging system possibly adaptable to similar systems is presented too.

gbvm has been subjected to an extensive series of experiments evaluating its performance both with regard to time and space performance. Two alternative policies for new train creation in the train algorithm have been investigated and our new train creation policy has been decided upon. A general purpose configuration has been vaguely suggested as a result of these experiments, but the important result is actually the difficulty in finding a suitable general compromise.

The speed of the write barrier has been quantified to lie below 10% of the total execution time on average. Compared to gbeta, gbvm is about 15 times faster, while gbetai is about twice as fast as gbvm with the set of benchmarks run.

Finally, suggestions for improving the performance of both the virtual machine component and the memory manager have been presented. The virtual machine could use a two-pass parser which would allow for a number of simple optimizations. The memory manager could be made adaptive in its moment and frequency of scavenging using various strategies for approximating the amount of dead bytes in the train space.

A. Test Programs

allocator.gb

```
-- betaenv:descriptor --
(#
t: @integer;
x: (# do 1+1->t; #);
y: ^x;
do
  (for 800 repeat
  &x[]->y[];
   (for 200 repeat
y[];
x;
   for);
   for);
#)
```

derive.gb

```
-- betaenv:descriptor --
(#
    derive: (#
     result: @real;
     do
     (for i:50000 repeat
     1*2+2*3+4*3+4*5+6*7+8*9+10*11->result;
     result+1.3*2.2+2.1*3.0+4.9*3.8+4.7*5.6+
     6.5*7.4+8.3*9.2+10.1*11.0->result;
     result+21.3*32.2+52.1*43.0+64.9*3.8+74.7*85.6+
     96.5*107.114+128.3*139.2+1410.1*1511.0+
     21.3*32.2+52.1*43.0+64.9*3.8+74.7*85.6+
     96.5*107.114+128.3*139.2+1410.1*1511.0+
     1.3*2.2+2.1*3.0+4.9*3.8+4.7*5.6+
     6.5*7.4+8.3*9.2+10.1*11.0->result;

     for);
    #);
    do
    derive;
 #)
```

cruncher3.gb

```
-- betaenv:descriptor --
(#
   list:
     (# element:< object;
        scan:
          (# current: ^element; c: ^cell
          do head->c[];
             (while c[]<>NONE do
                    c.elm[]->current[]; INNER; c.next[]->c[];
              while)
          #);
        add:
          (# c: ^cell enter (&c).elm[]
          do (if elements=0 then
                 c->head[]->tail[]; 1->elements
              else
                 c->tail.next[]->tail[]; (elements+1)->elements
             if)
          #);
        add3: (# enter (add,add,add) #);
        makeEmpty: (# do 0->elements; tail[]->head[] #);
        isEmpty: (# exit (elements=0) #);
        cell: (# elm: ^element; next: ^cell exit this(cell)[] #);
        head,tail: ^cell;
        elements: @integer
     exit this(list)[]
     #);

   node:
     (# child1,child2,child3: ^node; id: @string; myMark: @boolean;
        init: (# enter id exit this(node)[] #);
        childMethod:
          (# on1:< object; on2:< object; on3:< object;
             n: ^node; i: @integer
          enter i do (if i //1 then on1 //2 then on2 //3 then on3 if)
          #);
        changeChildN: childMethod
          (# on1::(# do n->child1[] #);
             on2::(# do n->child2[] #);
             on3::(# do n->child3[] #)
          enter n[] exit n
          #);
        getChild: childMethod
          (# on1::(# do child1->n[] #);
             on2::(# do child2->n[] #);
             on3::(# do child3->n[] #)
          exit n
          #);
        setId: (# enter id #);
        printMe:
          (# pr: (# n: ^node enter n[] do (if n[]<>NONE then n.printMe if)#)
          do id->stdio; child1[]->pr; child2[]->pr; child3[]->pr
          #);
        mark: (# do true->myMark #);
        unmark: (# do false->myMark #);
        isMarked:
          (# n: ^node; result: @boolean
          enter getChild->n[]
          do true->result;
             (if n<>NONE then n.myMark->result if)
          exit result
          #);
        makeChildren:
          (#
          do 'a'->node.init->child1[];
```

```
             ’b’->node.init->child2[];
             ’c’->node.init->child3[]
          exit (child1[],child2[],child3[])
  #)
     do (*id->stdio;*)
        INNER
     exit this(node)[]
     #);

   dfs:
     (# n: ˆnode enter n[]
     do n.mark;
(* n.id->stdio;*)
(*        n.printMe;*)
        (for i:3 repeat (if i->n.isMarked then i->n.getChild->dfs if)for)
     #);

   makeTree:
     (# nodeList: list(# element::node #);
        depth: @integer;
        listOld,listNew: ˆnodeList
     enter depth
     do ’root ’->node.init->root[]->(&listOld).add;
        (for i:depth repeat
             &listNew;
             listOld.scan(# do current.makeChildren->listNew.add3 #);
             listOld.makeEmpty;
             listNew->listOld[]
        for)
     exit root[]
     #);

   root,n1,n2,n3: ˆnode
do
   7->makeTree->root[];
(*  root.printMe;*)
   (for i:3 repeat root->dfs; (*’
n’->stdio*) for)
#)
```

constAlloc.gb

```
constAlloc.gb
-- betaenv:descriptor --
(#
   list:
     (# element:< object;
        scan:
          (# current: ^element; c: ^cell
          do head->c[];
             (while c[]<>NONE do
                    c.elm[]->current[]; INNER; c.next[]->c[];
              while)
          #);
        add:
          (# c: ^cell enter (&c).elm[]
          do (if elements=0 then
                 c->head[]->tail[]; 1->elements
              else
                 c->tail.next[]->tail[]; (elements+1)->elements
             if)
          #);
        add3: (# enter (add,add,add) #);
        makeEmpty: (# do 0->elements; tail[]->head[] #);
        isEmpty: (# exit (elements=0) #);
        cell: (# elm: ^element; next: ^cell exit this(cell)[] #);
        head,tail: ^cell;
        elements: @integer
     exit this(list)[]
     #);

   node:
     (# child1,child2,child3: ^node; id: @string; myMark: @boolean;
        init: (# enter id exit this(node)[] #);
        childMethod:
          (# on1:< object; on2:< object; on3:< object;
             n: ^node; i: @integer
          enter i do (if i //1 then on1 //2 then on2 //3 then on3 if)
          #);
        changeChildN: childMethod
          (# on1::(# do n->child1[] #);
             on2::(# do n->child2[] #);
             on3::(# do n->child3[] #)
          enter n[] exit n
          #);
        getChild: childMethod
          (# on1::(# do child1->n[] #);
             on2::(# do child2->n[] #);
             on3::(# do child3->n[] #)
          exit n
          #);
        setId: (# enter id #);
        printMe:
          (# pr: (# n: ^node enter n[] do (if n[]<>NONE then n.printMe if)#)
          do (*id->stdio;*) child1[]->pr; child2[]->pr; child3[]->pr
          #);
        mark: (# do true->myMark #);
        unmark: (# do false->myMark #);
        isMarked:
          (# n: ^node; result: @boolean
          enter getChild->n[]
          do true->result;
             (if n<>NONE then n.myMark->result if)
          exit result
          #);
        makeChildren:
          (#
          do ’a’->node.init->child1[];
             ’b’->node.init->child2[];
             ’c’->node.init->child3[]
          exit (child1[],child2[],child3[])
  #)
     do (*id->stdio;*)
        INNER
     exit this(node)[]
     #);

   dfs:
     (# n: ^node enter n[]
     do n.mark;
        n.printMe;
        (for i:3 repeat (if i->n.isMarked then i->n.getChild->dfs if)for)
     #);

   makeTree:
     (# nodeList: list(# element::node #);
        depth: @integer;
        listOld,listNew: ^nodeList
     enter depth
     do ’root ’->node.init->root[]->(&listOld).add;
        (for i:depth repeat
             &listNew;
             listOld.scan(# do current.makeChildren->listNew.add3 #);
             listOld.makeEmpty;
             listNew->listOld[]
        for)
     exit root[]
     #);

   root,n1,n2,n3: ^node
do
   7->makeTree->root[];
   3->root.getChild->n3[];
   3->n3.getChild->n1[];
   (for i:10 repeat root->dfs;
4->makeTree->n2[];
(2,n2[])->n1.changeChildN;
(*’
n’->stdio*)
for)
#)
```

dfstree.gb

```
-- betaenv:descriptor --
(#
   list:
     (# element:< object;
        scan:
          (# current: ^element; c: ^cell
          do head->c[];
             (while c[]<>NONE do
                    c.elm[]->current[]; INNER; c.next[]->c[];
              while)
          #);
        add:
          (# c: ^cell enter (&c).elm[]
          do (if elements=0 then
                 c->head[]->tail[]; 1->elements
              else
                 c->tail.next[]->tail[]; (elements+1)->elements
             if)
          #);
        add3: (# enter (add,add,add) #);
        makeEmpty: (# do 0->elements; tail[]->head[] #);
        isEmpty: (# exit (elements=0) #);
        cell: (# elm: ^element; next: ^cell exit this(cell)[] #);
        head,tail: ^cell;
        elements: @integer
     exit this(list)[]
     #);

   node:
     (# child1,child2,child3: ^node; id: @string; myMark: @boolean;
        init: (# enter id exit this(node)[] #);
        childMethod:
          (# on1:< object; on2:< object; on3:< object;
             n: ^node; i: @integer
          enter i do (if i //1 then on1 //2 then on2 //3 then on3 if)
          #);
        changeChildN: childMethod
          (# on1::(# do n->child1[] #);
             on2::(# do n->child2[] #);
             on3::(# do n->child3[] #)
          enter n[] exit n
          #);
        getChild: childMethod
          (# on1::(# do child1->n[] #);
             on2::(# do child2->n[] #);
             on3::(# do child3->n[] #)
          exit n
          #);
        setId: (# enter id #);
        printMe:
          (# pr: (# n: ^node enter n[] do (if n[]<>NONE then n.printMe if)#)
          do id->stdio; child1[]->pr; child2[]->pr; child3[]->pr
          #);
        mark: (# do true->myMark #);
        unmark: (# do false->myMark #);
        isMarked:
          (# n: ^node; result: @boolean
          enter getChild->n[]
          do true->result;
             (if n[]<>NONE then n.myMark->result else 'isMarked.n[] is NONE'->stdio if)
          exit result
          #);
        makeChildren:
          (#
          do 'a'->node.init->child1[];
```

```
             'b'->node.init->child2[];
             'c'->node.init->child3[]
          exit (child1[],child2[],child3[])
  #)
     do (*id->stdio;*)
        INNER
     exit this(node)[]
     #);

   dfs:
     (# n: ^node enter n[]
     do n.mark;
(*n.id->stdio;*)
        (*n.printMe;*)
        (for i:3 repeat (if i->n.isMarked then i->n.getChild->dfs if)for)
     #);

   wantedDepth: @integer;
   rememberRoot: ^node;

   makeTreedfs:
     (# depth: @integer;
id : @string;
n: ^node
enter (depth,id)
do
  id->node.init->n[];
  (if (depth <> 0) then
    ((depth-1),'a')->makeTreedfs->n.child1[];
    ((depth-1),'b')->makeTreedfs->n.child2[];
    ((depth-1),'c')->makeTreedfs->n.child3[];
  if)
exit n[]
     #);


   makeTree:
     (# nodeList: list(# element::node #);
        depth: @integer;
        listOld,listNew: ^nodeList
     enter depth
     do 'root '->node.init->root[]->(&listOld).add;
        (for i:depth repeat
             &listNew;
             listOld.scan(# do current.makeChildren->listNew.add3 #);
             listOld.makeEmpty;
             listNew->listOld[]
        for)
     exit root[]
     #);

   root,n1,n2,n3: ^node
do
   'root '->node.init->root[];
(*   5->wantedDepth;*)
   (7,'root')->makeTreedfs->root[];
(*   3->makeTree->root[]; *)
   (for i:3 repeat root->dfs; (*'
n'->stdio*) for);
(*   root.printMe;*)
#)
```

cruncher3JAVA.gb

```
-- betaenv:descriptor --
(#
   list:
     (# element: node;
        scan:
          (# current: ^element; c: ^cell; listNew: ^list;
  enter listNew[]
          do head->c[];
             (while c[]<>NONE do
                    c.elm[]->current[];
    current.makeChildren->listNew.add3;
            c.next[]->c[];
              while)
          #);
        add:
          (# c: ^cell enter (&c).elm[]
          do (if elements=0 then
                   c->head[]->tail[]; 1->elements
                else
                   c->tail.next[]->tail[]; (elements+1)->elements
             if)
          #);
        add3: (# enter (add,add,add) #);
        makeEmpty: (# do 0->elements; tail[]->head[] #);
        isEmpty: (# exit (elements=0) #);
        cell: (# elm: ^element; next: ^cell exit this(cell)[] #);
        head,tail: ^cell;
        elements: @integer
     exit this(list)[]
     #);

   node:
     (# child1,child2,child3: ^node; id: @string; myMark: @boolean;
        init: (# enter id exit this(node)[] #);
        childMethod:
          (# on1:< object; on2:< object; on3:< object;
             n: ^node; i: @integer

          #);
        changeChildN: (# n: ^node; i: @integer;
          enter (i, n[])
  do (if i //1 then n->child1[] //2 then n->child2[] //3 then n->child3[] if)
          exit n
          #);
        getChild: (# n: ^node; i: @integer;
          enter (i, n[])
  do (if i //1 then child1->n[] //2 then child2->n[] //3 then child3->n[] if)
          exit n
          #);
        setId: (# enter id #);
        printMe:
          (# pr: (# n: ^node enter n[] do (if n[]<>NONE then n.printMe if)#)
          do (*id->stdio;*) child1[]->pr; child2[]->pr; child3[]->pr
          #);
        mark: (# do true->myMark #);
        unmark: (# do false->myMark #);
        isMarked:
          (# n: ^node; result: @boolean
          enter getChild->n[]
          do true->result;
             (if n<>NONE then n.myMark->result if)
          exit result
          #);
        makeChildren:
          (#
          do ’a’->node.init->child1[];
             ’b’->node.init->child2[];
             ’c’->node.init->child3[]
          exit (child1[],child2[],child3[])
  #)
     do (*id->stdio;*)
        INNER
     exit this(node)[]
     #);

   dfs:
     (# n: ˆnode enter n[]
     do n.mark;
        (* n.printMe; *)
        (for i:3 repeat (if (i,n[])->n.isMarked then (i,n[])->n.getChild->dfs if)for)
     #);

   makeTree:
     (# depth: @integer;
        listOld,listNew: ˆlist
     enter depth
     do ’root ’->node.init->root[]->(&listOld).add;
        (for i:depth repeat
             &listNew;
             listNew[]->listOld.scan;
             listOld.makeEmpty;
             listNew->listOld[]
        for)
     exit root[]
     #);

   root,n1,n2,n3: ˆnode
do
   5->makeTree->root[];
   (for i:3 repeat root->dfs; (*’
n’->stdio*) for)
#)
```

cruncher3.java

```
class List{

    class Cell {
        public Node elm;
        public Cell next;

        public Cell(){
        }
    }

    Cell head = null;
    Cell tail = null;
    int elements = 0;

    Cell c = null;
    Node current = null;

    public void scan(List listNew){
        c = head;
            while(c != null){
                current = c.elm;
                current.makeChildren();
                listNew.add3(current.child1, current.child2, current.child3);
                c = c.next;
            }
        }

    public void add(Node elm){
        Cell c = new Cell();
        c.elm = elm;
        if(elements == 0){
            head = c;
            tail = c;
            elements = 1;
        } else {
            tail.next = c;
            tail = tail.next;
            elements++;
        }
    }

    public void add3(Node elm1, Node elm2, Node elm3){
        add(elm1);
        add(elm2);
        add(elm3);
    }

    public void makeEmpty(){
        elements = 0;
        tail = head;
    }

    public boolean isEmpty(){
        if(elements==0){
            return true;
        } else {
            return false;
        }
    }
}

class Node{
    Node child1,child2,child3;
    String id;
    boolean myMark;
```

```
public Node(String str){
    id = str;
}

public Node changeChildN(int i, Node n){
    if (i == 1){
        child1 = n;
    } else if(i == 2) {
        child2 = n;
    } else if(i == 3) {
        child3 = n;
    }
    return n;
}

public Node getChild(int i){
    if (i == 1){
        return child1;
    } else if(i == 2) {
        return child2;
    } else if(i == 3) {
        return child3;
    }
    return null;
}

public void setId(String _id){
    id = _id;
}

public void printMe(Node n){
    if(n != null){
        if(child1 != null){
            child1.printMe(child1);
        }
        if(child2 != null){
            child2.printMe(child2);
        }
        if(child3 != null){
            child3.printMe(child3);
        }
    }
}

public void mark(){
    myMark = true;
}

public void unMark(){
    myMark = false;
}

public boolean isMarked(int i){
    Node n = getChild(i);
    boolean result;
    result = true;
    if(n != null){
        result = n.myMark;
    }
    return result;
}

public Node makeChildren() {
    child1 = new Node("a");
    child2 = new Node("b");
    child3 = new Node("c");
```

```
        return this;
    }
}

class Main{

    Node root, n1, n2, n3;

    public void dfs(Node n){
        n.mark();
        for(int i = 1; i < 4; i++){
            if(n.isMarked(i)){
                dfs(n.getChild(i));
            }
        }
    }

    public Node makeTree(int depth){
        List listOld = new List();
        List listNew;
        Node root = new Node("root ");
        listOld.add(root);
        for(int i = 1; i < depth ; i++){
            listNew = new List();
            listOld.scan(listNew);
            listOld.makeEmpty();
            listOld = listNew;
        }
        return root;
    }

    public void start(){
        root = makeTree(5);
        for(int i = 1; i<4; i++){
            dfs(root);
        }
    }

    public static void main(String args[]){
        Main m = new Main();
        m.start();
    }
}
```

simple.gb

```
-- betaenv:descriptor --
(#
t: @integer;
x: (# do 1+1->t; #);
do
  (for 1000 repeat
  x;
   for);
#)
```

B. The Test Machine

CPU

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 5
model           : 2
model name      : Pentium 75 - 200
stepping        : 12
cpu MHz         : 133.270677
fdiv_bug        : no
hlt_bug         : no
sep_bug         : no
f00f_bug        : yes
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 1
wp              : yes
flags           : fpu vme de pse tsc msr mce cx8
bogomips        : 53.04
```

Memory

```
64MB
```

Operating System

```
Linux version 2.2.14-5.0 (root@porky.devel.redhat.com)
(gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release))
#1 Tue Mar 7 20:53:41 EST 2000
```

Compiler

```
GNU c++ version 2.95.3 with i586-pc-linux-gnu target machine
```

PCI-devices

```
PCI devices found:
  Bus  0, device   0, function  0:
    Host bridge: Intel 82439HX Triton II (rev 3).
      Medium devsel.  Master Capable.  Latency=32.
  Bus  0, device   7, function  0:
    ISA bridge: Intel 82371SB PIIX3 ISA (rev 1).
      Medium devsel.  Fast back-to-back capable.  Master Capable.  No bursts.
  Bus  0, device   7, function  1:
    IDE interface: Intel 82371SB PIIX3 IDE (rev 0).
      Medium devsel.  Fast back-to-back capable.  Master Capable.  Latency=32.
      I/O at 0xe800 [0xe801].
  Bus  0, device  11, function  0:
    Ethernet controller: 3Com 3C905 100bTX (rev 0).
      Medium devsel.  IRQ 11.  Master Capable.  Latency=32.  Min Gnt=3.Max Lat=8.
      I/O at 0xe000 [0xe001].
  Bus  0, device  12, function  0:
    VGA compatible controller: ATI Mach64 VT (rev 64).
      Medium devsel.  Fast back-to-back capable.
      Non-prefetchable 32 bit memory at 0xfa000000 [0xfa000000].
      I/O at 0xd800 [0xd801].
```

C. Graphs

Bibliography

[App98] Andrew W. Appel. *Modern Compiler Implemetation*. Cambridge University Press, 1998. ISBN 0-521-58388-8.

[Arm98] Eric Armstrong. Hotspot: A new breed of virtual machine. *JavaWorld*, March 1998. `http://www.javaworld.com/javaworld/jw-03-1998/jw-03-hotspot.html`.

[BC95] Lee Braine and Chris Clack. The importance of being lazy. June 1995.

[BG00] Lars Bak and Steffen Grarup. Automatic memory management in the java^{TM} programming language, 2000. `http://www.daimi.aau.dk/~larsbak/12-9/slides.pdf`.

[CFM$^+$97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Maio Wolczko. Compiling java just in time. *IEEE Micro*, 17(3):36–43, May-June 1997.

[Cor01] Rational Software Corporation. Rational purify for unix v2001a. `http://www.rational.com/products/purify_unix/index.jsp`, 2001.

[CWB86] P. J. Caudill and A. Wirfs-Brock. A third generation Smalltalk-80 implementation. *ACM SIGPLAN Notices*, 21(11):119–130, November 1986.

[Eng99] Joshua Engel. *Programming for the Java Virtual Machine*, chapter 1. Addison-Wesley, Reading, MA, USA, 1999.

[Ern99] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[Ern00] Erik Ernst. gbeta tutorial. `http://www.daimi.au.dk/~eernst/gbeta/index_tutorial.html`, February 2000.

[FPEN94] Gene F Franklin, J. David Powell, and Abbas Emani-Naeini. *Feedback Control Of Dynamic Systems*. Addison Wesley Publishing Company, third edition, 1994.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. `gprof`: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80 – The Language*, chapter 21. Addison-Wesley, Reading (MA), 1989.

[GS93] Steffen Grarup and Jacob Seligmann. Incremental mature garbage collection. Master's thesis, Aarhus University, Computer Science Department, August 1993.

[HH93] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. *OOPSLA-gc*, 1993.

[HM92] "Richard L. Hudson and J. Eliot B. Moss". "incremental garbage collection for mature objects". In "Yves Bekkers and Jacques Cohen", editors, *"Proceedings of International Workshop on Memory Management"*, volume "637" of *"Lecture Notes in Computer Science"*, "St Malo, France", "16–18 " sep "1992". "Springer-Verlag". "University of Massachusetts, USA".

[HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. A comparative performance evaluation of write barrier implementations. *ACM SIGPLAN Notices*, 27(10):92–109, October 1992.

[Höl93] Urs Hölzle. A fast write barrier for generational garbage collectors. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993. `ftp://self.stanford.edu/pub/papers/write-barrier.ps.Z`.

[Hud00] Rick Hudson. Gc writer's guide for open runtime environment. `http://www.intel.com/mrl/orp/orp\_tgz.tgz`, September 2000.

[IP01] Peer Møller Ilsøe and Simon Hem Pedersen. *GBVM - a gbeta Virtual Machine*. Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7E, DK - 9220 Aalborg Øst, 1. edition, January 2001.

[JJW01] Ricki Jensen, Christian Jørgensen, and Michael Wojciechowski. gVM - a Stand-alone gbeta Virtual Machine. Masters thesis, Department of Computer Science, Institute for Electronic Systems, Aalborg University, Fredrik Bajers vej 7a1 9220 Aalborg Øst, Denmark, 2001.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection - Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996. ISBN 0-471-94148-4.

[LH83] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. `http://lieber.www.media.mit.edu/people/lieber/Liebary/GC/Realtime/Realt%ime.html`.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*, chapter 3, pages xvi + 475. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[Mos87] J. Eliot B. Moss. Managing stack frames in Smalltalk. *ACM SIGPLAN Notices*, 22(7):229–240, July 1987.

[Nel79] P. A. Nelson. A comparison of PASCAL intermediate languages. *ACM SIGPLAN Notices*, 14(8):208–213, August 1979.

[PB98] Kevin M. Passino and Kevin L. Burgess. *Stability Analysis of Discrete Event Systems*. Number ISBN 0-471-24185-7. John Wiley & Sons, Inc., 1998.

[Pro01] The GNU Project. Data display debugger v3.3. `http://www.gnu.org/software/ddd/`, 2001.

[SaCL00] Witawas Srisa-an, J. Morris Chang, and Chia-Tien Dan Lo. Do generational schemes improve the garbage collection efficiency? *IEEE*, pages 58–63, 2000. 0-7803-6418-X/00.

[Sag00] Ajit Sagar. Java code compilation - the "write once, compile anywhere" solution. *Java Developers Journal*, 2000. `http://www.sys-con.com/java/archives/0401/sagar/index_c.html`.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. Number 0-534-94728-X. PWS Publishing Company, 1997.

[SM01] Inc. Sun Microsystems. `hotspot2_0-src-win/src/share/vm/memory/*.hpp`. webpage: `http://java.sun.com/products/hotspot/` or `http://www.sun.com/software/communitysource/hotspot/download.html`, May 2001.

[Sta97] William Stallings. *Operating Systems - Internals and Design Principles*. Prentice Hall, third edition, 1997.

[SUH86] A. D. Samples, D. Ungar, and P. Hilfinger. SOAR: Smalltalk without bytecodes. *ACM SIGPLAN Notices*, 21(11):107–118, November 1986.

[UJ88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. *ACM SIGPLAN Notices*, 23(11):1–17, November 1988.

[Ven96] Bill Venners. Under the hood: The lean, mean, virtual machine. *Javaworld*, 1996. `http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html`.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. *Bekkers and Cohen IWMM92*, 1992. `ftp://ftp.cs.utexas.edu/pub/garbage/gcsurv.ps`.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`.

[WM89] Paul R. Wilson and Thomas G. Moher. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, May 1989.

[YMP$^+$99] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. Latte: a java vm just-in-time compiler with fast and efficient register allocation. *Internation Conference on Parallel Architectures and Compilation Techniques, Proceedings*, pages 128–138, May-June 1999.

[ZG92] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Notices*, 27(12):71–80, December 1992.

Index