University of Manchester
School of Computer Science
Project Report 2012

Generational Garbage Collection in the Maxine Virtual Machine

Author: Andrei Voiculet
BSc(Hons) Computer Science

Supervisor: Dr. Mikel Luján

Abstract

Generational Garbage Collection in the
Maxine Virtual Machine

Author: Andrei Voiculet
BSc(Hons) Computer Science

The generational garbage collector technique is based on the weak generational hypothesis, which states that objects become garbage shortly after their allocation. The aim of this project is to learn about the internal structure of garbage collection and to implement a generational garbage collector. The Maxine Virtual Machine, in which the current state of the art in terms of garbage collection is a single threaded semispace collector, has been chosen as the development platform for this garbage collector.
This report presents information regarding the need for automating memory management and the motivation behind this project. Furthermore, it outlines the background information gathered before the start of the development, establishing the benefits and drawbacks of the fundamental techniques compared to the generational approach. General information regarding the structure of the Maxine Virtual Machine is given along with the design and implementation details of the newly developed garbage collector. In addition, the testing methodology is presented and the various experiments performed are discussed. Finally, a conclusion of the work undertook is provided and suggestions for future improvements are proposed.

Supervisor: Dr. Mikel Luján

Acknowledgements

I would like to thank my supervisor Dr. Mikel Luján who was always there when I needed his guidance. The amount of time that he has dedicated along with his patience has been impressive. Also I would like to thank Dr. Laurent Daynes for his invaluable advice on tackling certain difficulties regarding the Maxine Virtual Machine and the rest of the Maxine team.

In addition, I would like to thank Miruna Hanu for her support. Thanks also go to my parents who have supported me during this university experience and without whom I would not be where I am at the moment.

Last but not least, I would like to thank Radu Chiorean and Ioan Emanuel Ciocoiu for criticising this report.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Historically, memory management was performed explicitly by the software developer. Languages such as C or C++ require the programmer to allocate and free memory after it is no longer in use. This approach can give rise to a number of problems which are solved by automating this process. Modern programming languages incorporate this technique in the form of garbage collection and enforce this idea by not allowing the programmer to manage his own memory resources. The standard Java Virtual Machine (JVM), which runs applications implemented in the Java programming language, uses a garbage collector to reclaim the memory associated with objects which are no longer in use by the application. The aim of this project was to learn about garbage collectors and to implement such a facility of automatic memory management. The generational garbage collector has been chosen to be implemented within the Maxine Virtual Machine, which currently has a single threaded semispace garbage collector. In addition, this report will generally focus on garbage collection in an object oriented environment, with an emphasis on Java.

1.1 Explicit Memory Management

Manual memory management introduces a number of programming mistakes with security and reliability implications. Amongst them are the occurrences of dangling references and space leaks. The dangling references may appear as the result of deallocating a memory resource to which another structure may still hold a pointer. After freeing that resource, that particular memory address may be allocated to another structure resulting in erroneous program behaviour. Space leaks are also a common error which may appear while performing explicit memory management. They may occur, for example, in the situation in which the programmer can deallocate the first element of a linked list by mistake, consequently making the other elements unadressable by the program. Therefore it would be impossible at this stage to free the memory associated with the rest of the elements of the list [3].

1.2 Implicit Memory Management

The problems caused by having explicit memory management are solved by performing this process implicitly using a *garbage collector*. In addition to ensuring the delivery of more reliable code than by manual managing the resources, the garbage collector also provides increased abstraction, by decoupling this task from the programmer. JavaScript and .Net Framework also use a garbage collector to manage their memory resources.

1.3 Motivation and Requirements

As the aim of this project is learning about garbage collection, the closest possible approach to the state of the art garbage collection techniques, feasible to achieve in an undergraduate final year project, has been chosen to be implemented. A form of the generational garbage collector is in use by the Hotspot Java Virtual Machine which is the production quality JVM owned by Oracle, therefore this kind of garbage collector was a good candidate for the project.

The main requirements for achieving the generational garbage collector have been identified:

- The heap should be partitioned in generations;
- The heap should provide thread safe allocation;
- The collector must correctly reclaim memory;
- There should be means for the garbage collection of all heap partitions;

All the concepts required to understand these requirements are described in later chapters.

1.4 Report Structure

The rest of this section provides an overview of the remaining chapters.

- **Chapter 2 - Background**

  This chapter presents relevant background information relating to the different approaches to garbage collection and the improvements that the generational approach can provide.

- **Chapter 3 - Maxine VM**

  The Maxine VM, the virtual machine used by this project as the development platform, is described in this chapter.

- **Chapter 4 - Design**

  This part of the report focuses on design information of the system implemented in this project.

- **Chapter 5 - Implementation**

  The description of implementation details of the components presented in the design chapter follows.

- **Chapter 6 - Testing and Evaluation**

  Evidence relating to the various methods of testing used is provided and various experiments done are discussed.

- **Chapter 7 - Conclusion**
  This chapter concludes the work done and provides insight regarding future work that could be undertaken.

Chapter 2

Background

Key terminology needs to be defined for the report to be fully understood. Furthermore, the fundamental techniques used for garbage collection are presented along with their advantages and drawbacks. Following this, the generational garbage collection technique, which this project has implemented, is presented and then put in contrast with these fundamental approaches. It is worth noting that most of the information contained in this chapter has been extracted from [4].

2.1 Terminology

The *heap* is a memory region in which objects are allocated. It can be partitioned to support certain implementations of garbage collection.

An object used by the application is allocated in the heap and is stored as contiguous bytes. It may contain references to other objects, or values such as integers or characters (primitive types in Java terminology). It is common that every object contains a header field to store meta-data such as the class it belongs to, and information about locks or hash code [4].

The program executing with garbage collection was broken down by Dijkstra *et al.* [5] in two actors: the *mutator* which would be responsible for the computation, including allocation of new objects or the altering of the object graph and the *collector* which executes garbage collection. These two terms will be used throughout this project to denote the separation between the running of the application and the garbage collection. The object graph can be constructed with its nodes represented by the objects and its edges by the object references. A traversal of this graph can also be refered to as *tracing*.

An object is considered *alive* if it is still used by the program and it should not be reclaimed. Liveness of an object can be determined by examining its reachability starting from a set of objects that are known to be alive, also known as the *mutator roots*, and then examining each reference of each visited object. These mutator roots represent pointers to objects directly accessible by the program and can be stored in the thread stacks, thread local variables or registers. Moreover, the root set is not constant, and will change throughout the program execution. Therefore, in order for an object to be alive, it must be reachable from the root set. An unreachable object is considered garbage, and it cannot become alive again [4].

Heap fragmentation represents the appearance of chunks of free memory between the cells (variable sized divisions of the heap) associated with objects allocated in the heap. Effectively, they can be thought of as holes in the heap.

For this project it has been assumed that garbage collection is performed on a single thread. However, there can be multiple mutating threads.

2.2 Garbage Collection

Garbage collection must be correct, thus it should never reclaim the memory associated with a live object. There are four fundamental techniques which are used for constructing all garbage collection routines: *mark-sweep collection, mark-compact collection, copying collection,* or *reference counting*. Certain garbage collectors, hybrids, may use different approaches to collect certain parts of the heap. All mutating threads are stopped at points in time considered safe to check the mutator roots, also known as "stopping the world", and then the collecting thread starts [4].

2.2.1 Mark-Sweep

The mark-sweep technique was the first used for automatic memory management [6]. The reclamation of memory is done in two phases: the *marking* phase in which the object graph is traversed and all reachable objects from the mutator roots are marked, and the *sweeping* phase in which the memory associated with unmarked objects, considered garbage, is reclaimed [4]. Figure 2.1 shows snapshots of the heap before the collection starts and after the mark phase and respectively sweep phase are completed. Grey accounts for free spaces in the heap, blue slots represent memory occupied by unmarked objects and orange shows cells containing marked objects.

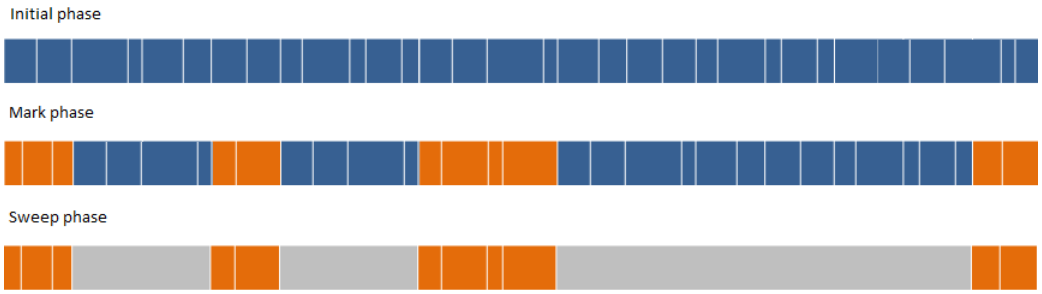


Figure 2.1: Heap snapshots during the mark-sweep garbage collector's phases.

Caches are designed based on the assumption that applications exhibit good temporal and spatial locality. Temporal locality refers to the property that a memory location recently accessed is likely to be accessed again shortly, while spatial locality means that a memory access implies that its neighbouring locations will also be accessed. Due to the fact that most objects are not referenced by more than one pointer, garbage collection accesses most objects once. For this reason, the mark-sweep garbage collection exhibits poor cache behaviour, in the sense that low benefit can be gained from the cache if values are only used a small number of times [4].

There are various techniques which may be employed to improve the performance of this garbage collector such as bitmap marking and lazy sweeping. The first technique implies the use of a bitmap table for storing mark bits rather than including them in the object header. A small memory overhead is incurred, but object headers do not need to be modified, thus

the bitmap marking is likely to invalidate fewer cache lines, therefore reducing the number of write-back to memory operations [4].

The idea behind the second technique is based on the fact that the death of an object is permanent. Taking this into account, it would be possible to employ a "sweeping" thread to run in parallel with the mutator threads to reclaim memory. A second possibility would be to incorporate the sweeping functionality inside the mutator. In this case, the memory occupied by an unmarked object would be freed whenever a new object allocation is performed. In addition, by employing this latter technique, the temporal locality will be improved because memory locations being collected will usually be used for object allocation shortly after [4].

Advantages of the mark-sweep approach include no mutator overhead in terms of read and write operations, unlike reference counting which has additional overhead associated with both kinds of operations. The use of the lazy sweeping technique gives good throughput (percentage of time spent outside garbage collection [3]). This collector also has good space usage compared to copying collectors, which will be presented in one of the upcoming sections [4]. On the other hand, there are certain problems associated with this collection routine such as heap fragmentation, which may lead to a considerably high amount of unusable memory. It also requires the use of more complicated allocators to find free locations in the heap to allocate in [4].

2.2.2 Mark - Compact

The mark-compact garbage collection algorithm is an improvement on the basic mark sweep counter part. As the name suggests, it compacts live objects in the heap to eliminate memory fragmentation. This garbage collection routine is performed in multiple phases, a marking phase and subsequent ones in which objects are compacted and references of all live objects to recently moved objects are updated. The most popular techniques for this approach are the arbitrary and sliding compaction. Although the arbitrary compaction is fast to execute, the aleatory ordering of objects during compaction would reduce the application throughput due to poor spatial locality [7]. This issue is solved by using sliding compaction which moves objects either at the start or at the end of the heap. By doing this, the order in which objects were allocated is preserved. Therefore, the impact on locality is not as significant [4].

Mark-compact collection may have a small memory overhead depending on the particular algorithm used, for example the overhead in the object headers for storing forwarding pointers to their new location. Furthermore, this collection technique eliminates fragmentation completely and is mainly used in JVMs for heap compaction. However, compaction does impact on the application throughput, due to the necessity of multiple scannings of objects in the heap, therefore increasing the collection pause times [4].

2.2.3 Semispace Collection

The semispace garbage collector [8] is different from the collectors previously presented; it is a copying collector. The main characteristic of a copying collector is that it copies live objects to a different location (e.g. a different heap) and afterwards discards the former. The semispace collector requires a partitioned heap layout, specifically the heap needs to be divided in two halves, creating a "from space" and a "to space". Allocation takes place in the to space, while the from space is kept empty. The actual garbage collection routine is triggered when the allocation space is full. As shown in Figure 2.2, the two semispaces are swapped and all live

objects are copied from one space to the other. At this point, it is safe to assume that the only objects remaining in the from space without forwarding references (to the new copied memory location) in their headers, are dead. Thus the from space can be safely discarded and allocation may resume in the to space. Orange cells account for live objects, blue slots are occupied by garbage and grey represents free heap space.

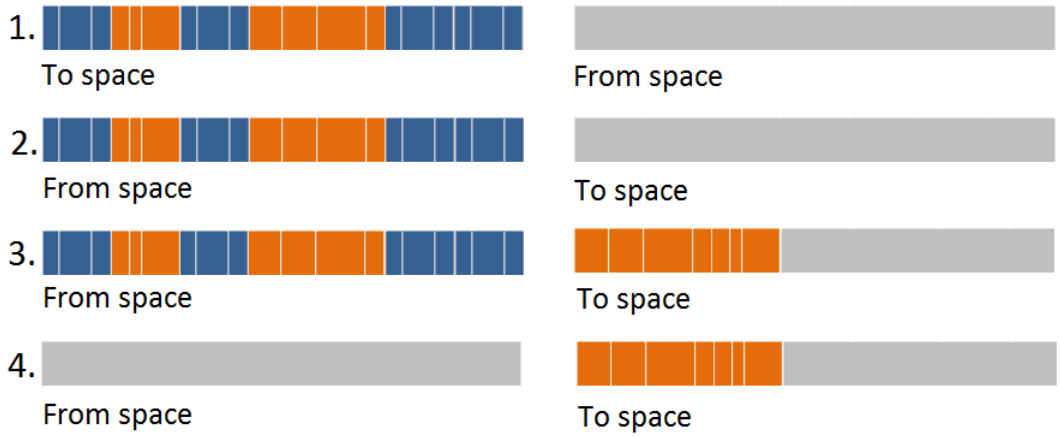


Figure 2.2: Heap Snapshots during a Semispace Garbage Collection.

Copying collection has the advantage of fast allocation, using a simple pointer increment technique. This type of allocation exhibits good cache behaviour because patterns in memory access can be anticipated and the hardware support for fetching data prior to its use, *prefetching*, can be taken advantage of [4]. Also, fragmentation is eliminated from the heap.

On the other hand, the inability to use half the memory available for the heap is a major drawback. As a consequence, there will be more collection cycles than in other collectors. In addition copying long lived large objects at each collection can lead to poor performance.

2.2.4 Reference Counting

The last fundamental approach to garbage collection is reference counting [9]. The techniques previously presented reclaim memory in an indirect fashion, the object graph is traversed for the discovery of live objects and the remaining unvisited ones are classified as garbage. In contrast, this collection algorithm detects dead objects by checking them directly. A reference count, usually stored in the object header, is associated with each object. If the reference count of an object reaches 0, that particular one is classified as garbage and its memory can be reclaimed immediately. Figure 2.3 illustrates the write operation performed by the mutator to replace a reference from an object's field with a new reference; the straight line arrow represents the newly created reference, while the dotted arrow represents the old one. This operation increments the reference counter of the newly created target object and decrements it for the old one. At the point where a reference count is 0, the memory associated with the corresponding object can be reclaimed and the counts of all objects referenced by the newly reclaimed one are decremented [4].

An immediate advantage of this approach is the fact that there is no need to "stop the world", because, as stated previously, the memory associated with an object can be reclaimed immediately after it becomes garbage. Reference counting also scales well with the size of the

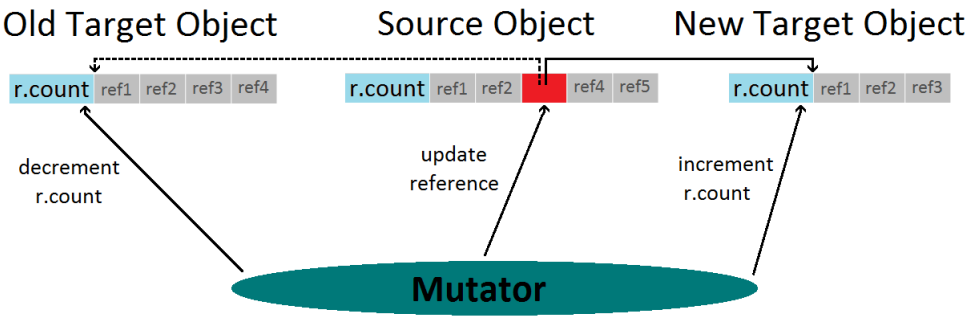


Figure 2.3: Reference count operation.

heap, because its cost is proportional to the number of pointer reads and writes, rather than the number of objects in the heap which is the case in the previously presented collectors [4].

However, the drawbacks of this approach outweigh the benefits. The most significant one would be that this collector is not able to reclaim the memory associated with cyclic data structures. Moreover, the mutator incurs significant overhead because all write as well as read operations bear the delay of updating the reference counts. A consequence of the latter would be that the updates will invalidate the cache. In addition, read-only operations modify the reference counts, therefore in a multi-threaded environment further synchronisation would be needed. Furthermore, the number of references to an object may be equal to the number of heap objects minus one. Taking into account that objects are small, particularly Java objects occupy 20 to 64 bytes [10], storing a count in an object header may incur a substantial memory cost [4].

2.2.5 Generational Garbage Collection

The generational garbage collector is a hybrid collector because it handles different parts of the heap in different ways. This technique exploits the weak generational hypothesis that most objects die young [11], namely they become garbage short after they have been allocated. This hypothesis is widely applicable regardless of programming paradigm or language. In addition, it has been observed that the longer an object may live, the less likely it is to die.

Taking this information into account, a generational garbage collector partitions the heap in two or more generations, typically a young generation (eden) and an old generation. Each generation may be managed by different algorithms. Usually, objects are allocated in the young generation and when it becomes full, a collection (minor collection) is triggered and all live objects are then copied to the old generation as presented in Figure 2.4. Orange represents heap slots with live objects, blue accounts for garbage and grey illustrates the free heap space. Notice the arrow from a dead object located in the old generation. It represents an intergenerational pointer and the figure illustrates how a reference from a dead object would promote other garbage. The objects in the old generation are further considered to be live and are not subject to tracing until the old generation fills up and needs to be collected (full collection). The collection of the old generation is typically done using one of the fundamental approaches: mark-sweep, mark-compact or semispace collection.

Collecting generations separately give rise to a problem, the issue of intergenerational pointers. As the old objects are not traced, there is a need for an alternative to determine which of them must be used to detect the live objects in the young generation. Generational

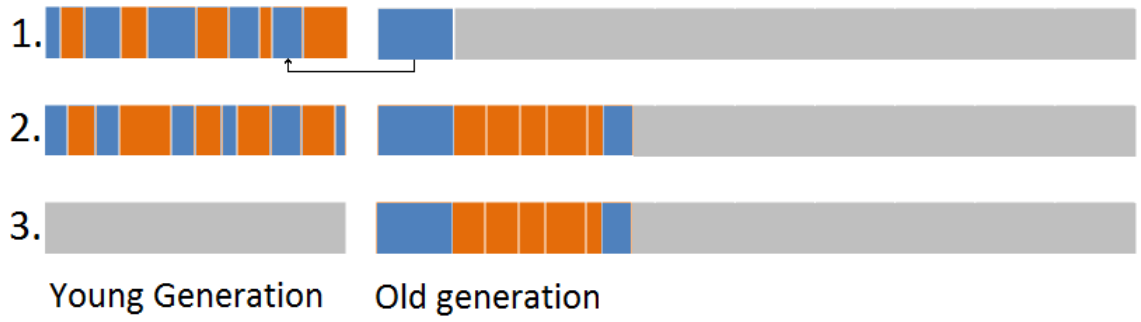


Figure 2.4: Heap snapshots during a minor collection of a generational garbage collector.

collectors impose an overhead on the mutator to detect the occurrences of pointers from the old to young objects and store the location of the object that may contain the intergenerational pointer, for future use. This kind of operation performed by the mutator is known as a write barrier. The time complexity of a minor collection is proportional to the number of live objects. Therefore a small allocation space, in which the objects might not have had time to die would result in an increased pause time. In addition, by not giving the objects the chance to die in eden, garbage will be created in the old generation which would lead to an increased number of time consuming full collections. It also promotes *nepotism*, because these newly tenured objects may become garbage quickly, but being considered roots of the old generation, they will enable the promotion of objects that may not have been copied otherwise (see Figure 2.4). Furthermore, new objects are modified more frequently than old ones, thus promoting objects quickly might impose significant overhead on the write barrier [4].

There are techniques which may be used to reduce the amount of early tenuring such as the addition of survivor spaces to the heap partitions, intermediate regions in which the objects are copied to allow them more time to become garbage before they are promoted. On the other hand, these improvements may introduce further overhead in pause times and are very complex to implement. In addition, the presence of several generations may impose a significant overhead on the write barrier due to the likelihood of having more intergenerational pointers [4].

2.2.6 Generational Garbage Collector vs other Collectors

In contrast with a pure mark-sweep approach, the generational collection does not suffer from memory fragmentation and has considerably lower pause times. While the mark-compact technique may improve the former with regard to memory fragmentation, it incurs higher pause times due to the costly heap compaction. Compared to the semispace garbage collector, the generational approach avoids repeatedly degrading performance by copying long lived objects and may also filter incoming garbage better than the former. Reference counting, could be desirable from the point of view of being able to reclaim memory as soon as an object dies, but it has a number of significant drawbacks which make it unsuitable for high performance systems.

Overall, generational garbage collectors may offer low pause times and by proper sizing of the young generation this can become unnoticeable in many environments. Furthermore, young generation allocation is done sequentially using a simple pointer increment. Based on this and on the fact that most writes are done on newly allocated objects, cache locality is obtained and the memory traffic may be reduced.

Generational garbage collection has been showed to be a ”highly effective organisation and to provide significant performance improvements over a wide range of applications” [4].

2.3 Summary

The fundamental approaches to garbage collection have been described and compared with the generational collector with the conclusion that the use of the latter would lead to a better performance, thus making it a reasonable candidate to demonstrate within a third year project.

Chapter 3

Maxine Virtual Machine

The Maxine Virtual Machine (Maxine VM) is a research purpose virtual machine. It is implemented in Java, the same programming language ran by the machine, justifying the title of being a meta-circular virtual machine. Moreover, its architecture is modular, therefore the independent development of different sub-components such as compilers, garbage collectors and their integration in the system is possible. Furthermore, the system profits from advanced language features present in Java 5, such as annotations, static imports and generics. In addition, the source code supports development in modern Integrated Development Environments (IDEs), which facilitates viewing of its large code base(548,482 lines of code excluding comments [12]) and the improvement of the code structure. Supported platforms include Solaris/x64, Open Solaris/x64, Linux/x64, Mac OS X/x64 [13].

3.1 Maxine VM vs Hotspot JVM

In contrast, the Hotspot Java Virtual Machine (Hotspot JVM) is Oracle's production quality JVM and it is implemented in C++. Although this language is object oriented, its code structure is less understandable than Java's. Also, it is present on the market since 1999 and it has been continuously improved since then. As the software's performance increases, the quality of the code tends to deteriorate, and may become unreadable for the outside community. In addition, the Maxine VM project is kept alive by Oracle for research purposes, for new components to be developed and tested quickly and to be afterwards integrated in the Hotspot JVM, e.g. the graal compiler, which is a new generation compiler, currently being integrated in Hotspot. Furthermore, the Hotspot JVM already benefits from a generational garbage collector, amongst other ones, whilst Maxine VM does not. For the reasons mentioned above, Maxine VM was chosen as the development environment for implementing a version of the generational garbage collector.

3.2 Boot Image

In order to start the virtual machine, a previously generated boot image is required. The boot image is a binary file which contains a memory image of the running virtual machine for a particular target platform. The boot image is generated by running the Maxine VM on the Hotspot JVM to the point where the VM can run on itself. Its contents include a populated heap, namely the boot heap and compiled code for the target platform [14]. The boot image is not an

executable, thus a small C program needs to be employed to start it. This process of starting the virtual machine on the boot image is called *bootstrapping*, the phase in which Maxine begins to execute itself. In addition, the boot image is loaded into memory, the Java main thread is created and objects contained in the image are mapped in memory, thus accessible via a memory address [15].

3.3 Threads

Threads in the Maxine VM are implemented using native threads scheduled by the operating system using a one to one mapping. Each thread may be in one of the following states: mutating, it may modify the contents of the heap by performing write operations or non-mutating state in which it does not perform any write operations on the heap.

The virtual machine natively allocates a contiguous memory region called a thread locals block (TLB) for each thread. During the boot image generation a contiguous block of memory, known as the thread local area (TLA), is allocated to hold a word corresponding to each thread local variable. Each variable is assigned an offset in the TLA, with the first location reserved for the *safepoint latch*. Three TLAs are assigned for each thread corresponding to one of the safepoint states in which the virtual machine can be: enabled, disabled, triggered and are placed within a TLB, as presented in Figure 3.1 [1].

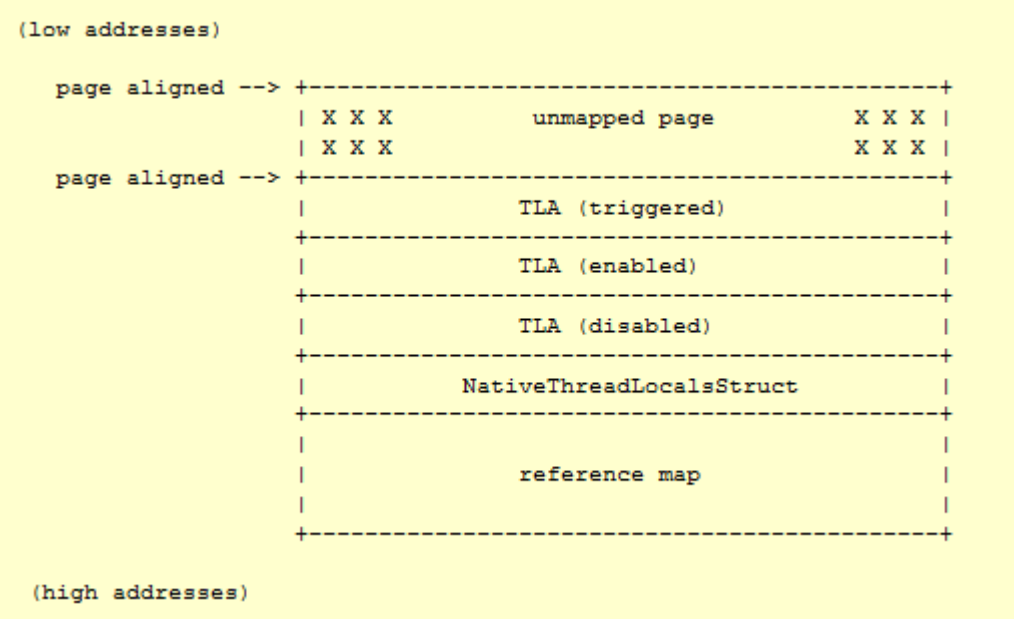


Figure 3.1: Thread Local Block's structure [1].

A safepoint is a special instruction (load instruction) in compiled VM code placed at a location where a thread can be frozen, providing consistency between the heap and the thread's stack; a safe place where a garbage collection may commence. These instructions may be placed within branches, switches or go to statements and their trigger causes a trap to the operating system. The first word of the TLA, the safepoint latch (representing the contents of register r14) contains the starting memory address of the TLA. If the thread is in enabled state, with regard to safepoints, and the special instruction is executed, the start address of the triggered TLA is loaded into the safepoint latch of the enabled TLA causing a trap to the

operating system. This particular kind of trap is handled by the VM and at this point it is safe for the virtual machine to start a garbage collection [16].

Furthermore, each thread has a stack reference map associated with it. This is a data structure used to identify the places within the thread's stack in which references to objects contained in the heap may be found. The reference map is prepared at a safepoint, after the threads have been frozen for a scheduled garbage collection. In addition the thread's stack reference map is also stored inside the TLB [1].

3.3.1 Thread Local Allocation Buffers

To avoid costly locking the heap to handle thread races, allocation is done using thread local allocation buffers (TLABs). These structures should not be confused with the previously presented TLAs or TLBs. These buffers are effectively portions of heap memory reserved, private for the requesting thread to allocate in. Therefore, there is no need for synchronization while the allocation takes place. Fast access to a TLAB belonging to a particular thread is provided through pointers held in the thread local variables of their TLAs. Also, when the TLAB is full, the thread may request another one to be allocated from the heap. Figure 3.2 shows TLABs allocated in the heap, grey accounting for free heap space, while the other different colours correspond to TLABs owned by different threads. The further partitioning of the allocation buffers account for objects which have been allocated so far.

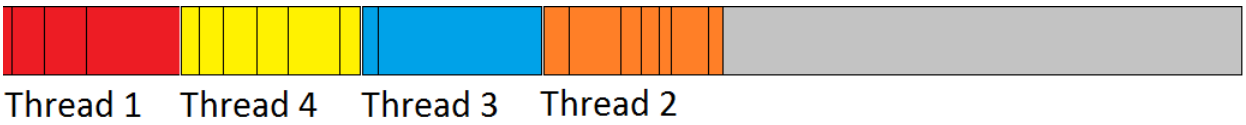


Figure 3.2: TLABs in the heap.

3.4 Object Representation

An object may be represented in the Maxine VM by three low-level memory representations: *tuples*, *arrays*, and *hybrids*.

A Maxine tuple is a memory representation used for Java instances. It contains a two word header and a collection of named values, called fields. Common for all object representations, the first word of the header is a pointer to the dynamic hub for the class. The second word, misc, is usually used to hold information regarding hash code and locks [2].

The array representation is used for Java array instances. In contrast to the former, it combines a three-word header and some fixed number of values of identical type. The first two words have the same contents as the tuple representation; the third word holds a value equivalent to the number of elements in the array [2].

The hybrid representation is used to represent Maxine hubs (the structures to which the first word of the object header refers to). It uses a three word header, similar to the array one, a collection of fields and an array of words [2].

Maxine hubs are hybrid instances that must be immediately accessible from the class instance. A hub is directly accessible from the object header, which, as specified previously, holds a pointer to it. Various information can be stored in a hub, including garbage collection related details such as the reference map or the memory representation of the current instance.

As the hubs themselves contain both arrays and named fields, they cannot be represented as Java objects, therefore they are represented as hybrids. Furthermore, dynamic hubs are pointed to by class instances and static tuples, by special kinds of tuples used by the VM internals [2].

Figure 3.3 illustrates the relationship between the object, the dynamic hub for the particular class instance, and the class actor for the class type. The latter is ,effectively, a representation of the implementation details of the Java class associated with the object. Also, the figure illustrates the memory representations of the various structures in the heap.

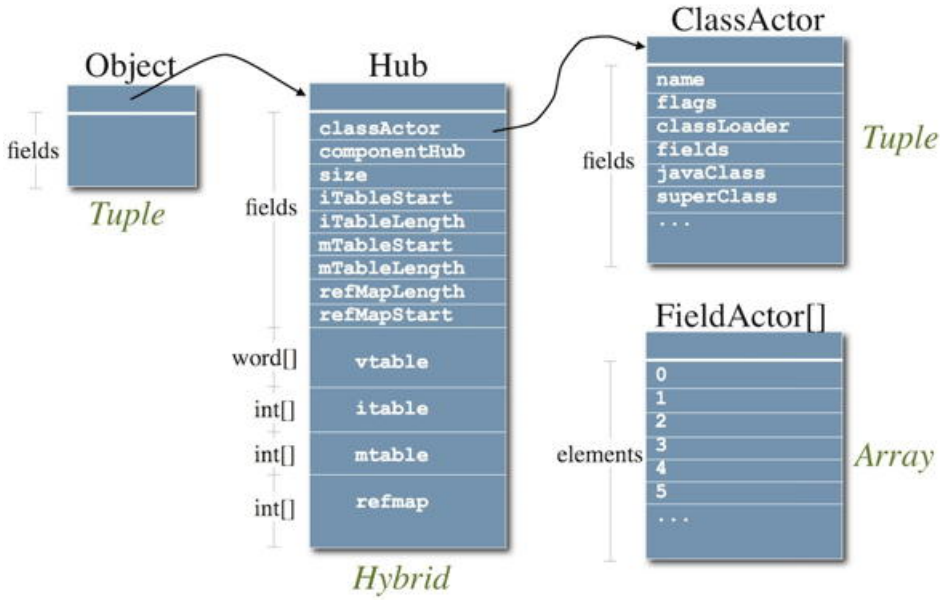


Figure 3.3: Relationship between Object-Hub-Class Actor [2].

3.5 Special References

Special references are all a form of weak references; such references to an object do not pull it to the live set. An object reachable only through a weak reference must be placed in a queue for further processing (finalization) performed by a reference handler thread (written in Java). Consequently, the object along with others it may reference must be kept alive.

For these reasons, special references demand special handling. If pointing to a dead object, it is kept alive, which in most cases requires another object graph traversal for the "ressurection" of all dead objects in the transitive closure of the special reference. The special reference handling mechanism is done by the SpecialReferenceManager class from the Maxine VM project.

3.6 Compiler

Maxine uses an optimising Just in time (JIT) compiler, C1X, derived from the Hotspot Client compiler (C1). The compiler is strictly separated from the runtime through a set of interfaces. C1X is separated from the runtime representation of classes, methods of a VM, by having a set of interfaces that specify only methods needed by the compiler in translating code. Similarly,

the runtime has the same characteristic, to be independent from the implementation details of the compiler using interfaces which define compiler output. In addition, it performs certain optimisations such as inlining or constant-folding [17].

3.7 Current Garbage Collector

The garbage collector currently present in the Maxine VM is a semispace garbage collector (see Section 2.2.3). However, it cannot usually attain efficient memory usage and good performance [11]. The Maxine VM team implemented this simple algorithm in order to make the VM runnable, thus to be able to concentrate on other priorities such as the compiler or JNI interface.

3.8 Card Table

A representation of the card table and write barrier is necessary in order to fully understand generational garbage collection. Both facilities are currently present in Maxine VM GC Toolkit.

The card table may be viewed as an array of bytes, each of its bytes (cards) corresponding to an area of the heap. Effectively, a logical partitioning of the heap in equally sized cards has been achieved.

The write barrier is executed when a write operation on a reference field of an object takes place. At this point, the card corresponding to the heap region where the modified object is located is dirtied. By doing this it is ensured that all object references from the old to the young generation can be detected at garbage collection by scanning all the heap regions corresponding to the dirty cards.

Chapter 4

System Design

Having presented the structure of the virtual machine, the design details of the heap scheme which supports generational garbage collection will be introduced. Firstly, an overview of the mechanism used for the integration of this component in the system is provided. A description of the architecture follows, presented using a class diagram to illustrate its full structure. In addition, the functionality of the heap scheme using generational garbage collection is presented in a top down fashion along with the core operations, allocations and collections.

4.1 Garbage Collector's Integration in the Maxine VM

Different components, as well as different implementations for the same component may exist as part of the Maxine VM. These components are represented in the VM by schemes implemented in specific classes i.e. HeapScheme, ReferenceScheme, LayoutScheme. However only one type of component may be binded to a VM scheme in a single run of the virtual machine, thus there was a need for a mechanism to specify the modules to be used by the VM at any given run. The Maxine VM implemented such a mechanism in the form of Maxine packages. This special kind of package is a collection of classes in a Java package which, in addition, contains a class named Package, providing the functionality of a marker to include the required scheme in the boot image. Figure 4.1 further illustrates the relationship between the components with regard to the Package mechanism. The VMConfigurator creates the configuration of the virtual machine using the package classes of the required schemes to be included (i.e. com.sun.max.vm.heap.sequential.generational.Package).

4.2 System Architecture

In order to fully understand the architecture of this system and its complexity, knowledge about all interactions between classes is required. Figure 4.3.1 illustrates the system class diagram to meet this requirement. Notice that the classes with a label reading "Maxine VM" represent classes already implemented in the virtual machine. The functionality contained in the other classes, implemented in this project, will be presented in the following sections.

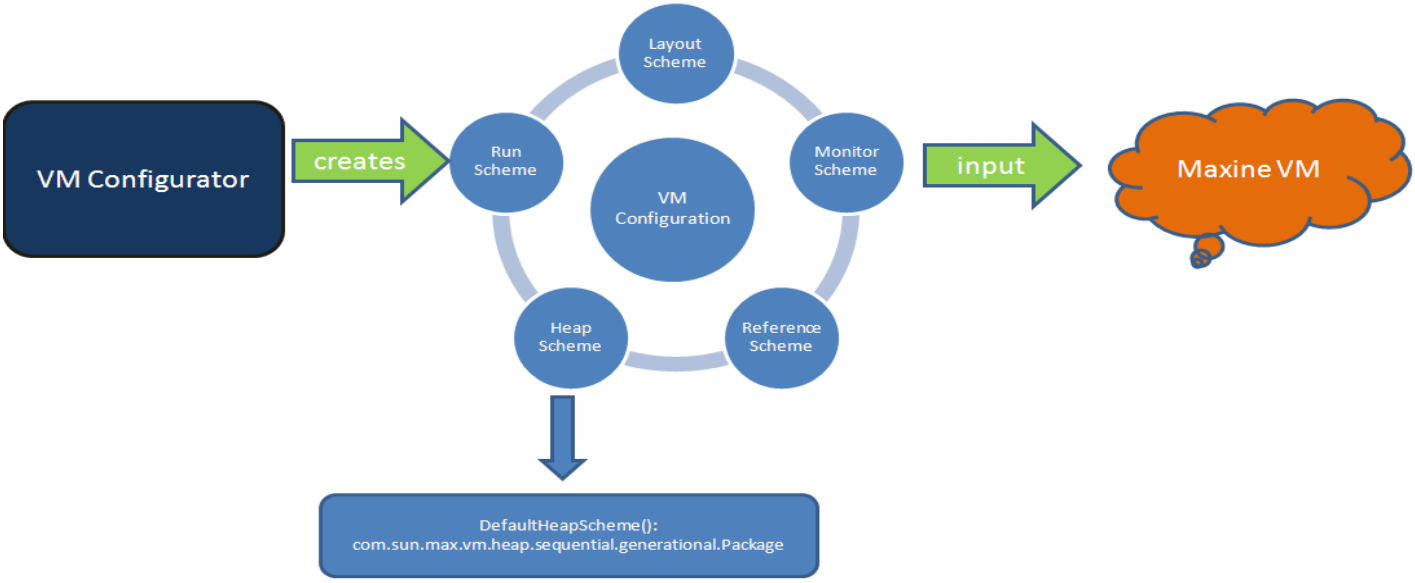


Figure 4.1: Heap Scheme Integration.

4.3 Generational Heap Scheme

The GenerationalHeapScheme class provides the heap layout and allocation related methods. It inherits the functionality common to all heap scheme implementations from the HeapSchemeWithTLAB class. This includes mechanisms for allocation through a TLAB, for the enabling or disabling of allocation or for the replacement of a TLAB when it becomes full.

4.3.1 Heap Layout

The heap is allocated as a contiguous memory region and in order to support generational garbage collection, it is partitioned in two generations: a young generation and an old generation. The latter is further divided in two semispaces to enable its collection by a semispace garbage collector. The heap along with the code cache and boot heap must be aligned in order to make use of the card table. The invariant of this remembered set dictates that it must cover the dynamic heap and in addition the boot heap and code cache. The reason is to avoid further write barrier overhead induced by unnecessary conditions. A side effect of this requirement is that the previously specified memory regions must be aligned with each other in order for the table to correctly record the memory locations in which an intergenerational pointer might have been created. Furthermore, the card table is allocated at the end of the allocated heap. Figure 4.2 illustrates the full memory layout.

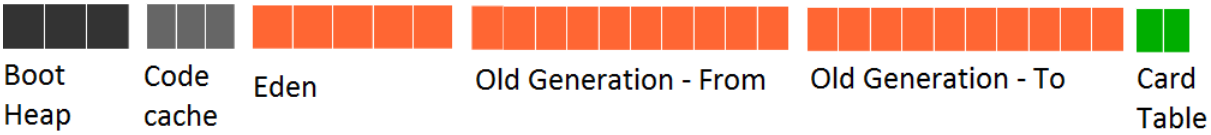
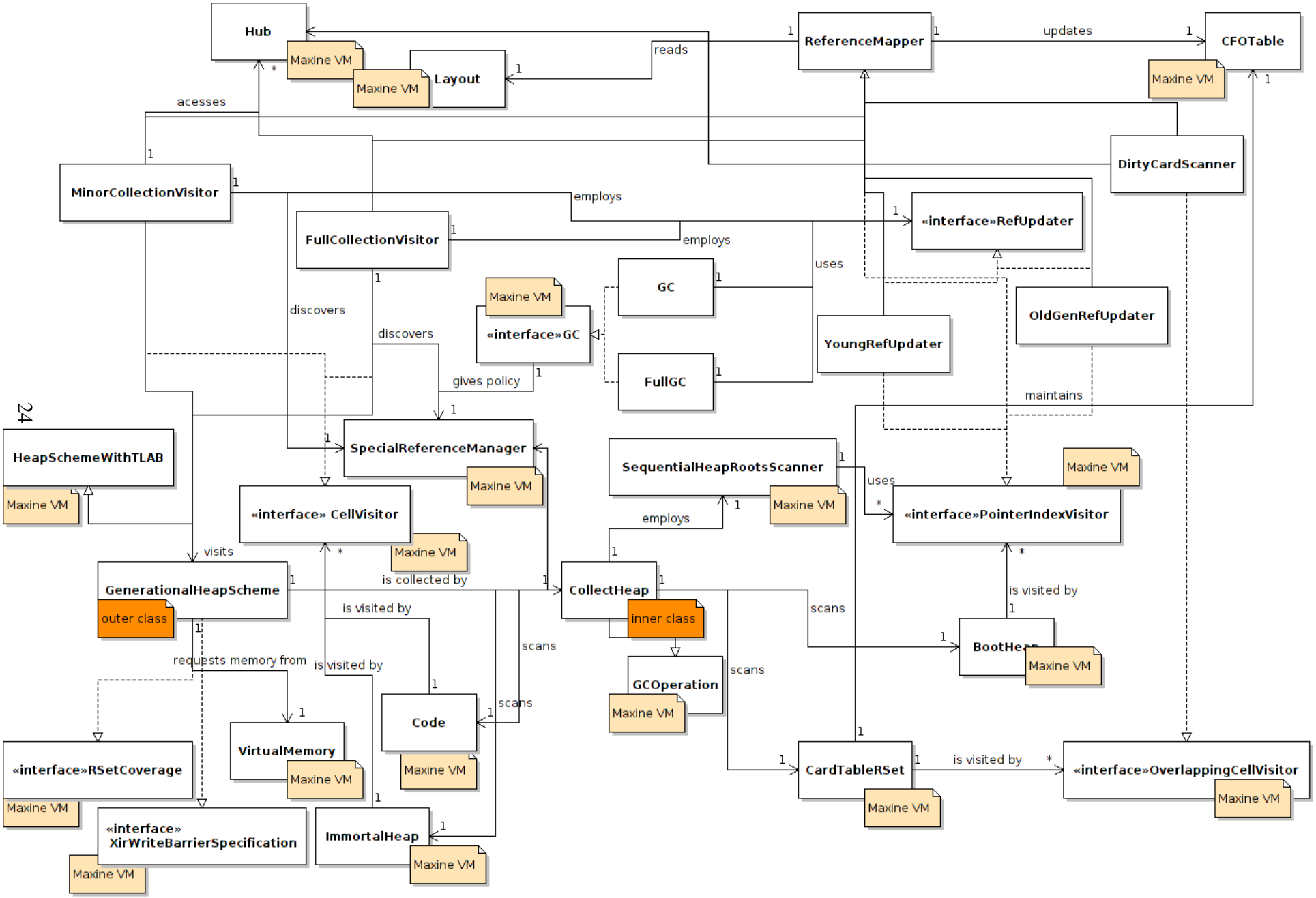


Figure 4.2: Memory Layout.


Hub
Maxine VM
Maxine VM
Layout
reads
ReferenceMapper
updates
CFOTable
Maxine VM
acesses
DirtyCardScanner
MinorCollectionVisitor
employs
«interface»RefUpdater
FullCollectionVisitor
employs
uses
GC
Maxine VM
«interface»GC
discovers
discovers
YoungRefUpdater
OldGenRefUpdater
FullGC
gives policy
maintains
HeapSchemeWithTLAB
Maxine VM
SpecialReferenceManager
Maxine VM
SequentialHeapRootsScanner
Maxine VM
uses
Maxine VM
«interface»PointerIndexVisitor
«interface» CellVisitor
Maxine VM
employs
visits
GenerationalHeapScheme
outer class
is collected by
CollectHeap
inner class
is visited by
is visited by
scans
BootHeap
Maxine VM
requests memory from
is visited by
scans
GCOperation
Maxine VM
scans
Code
Maxine VM
scans
VirtualMemory
Maxine VM
«interface»RSetCoverage
Maxine VM
CardTableRSet
Maxine VM
is visited by
«interface»OverlappingCellVisitor
Maxine VM
«interface»
XirWriteBarrierSpecification
Maxine VM
ImmortalHeap
Maxine VM

4.3.2 Allocation

Allocation of most new objects takes place in the young generation. It can be performed either directly from the heap or by using thread local allocation buffers (TLABs). Multiple threads may request the allocation of a TLAB from the heap, therefore synchronization is needed to avoid thread races. The heap does not suffer of fragmentation, thus the allocation may proceed sequentially without the need of other policies for finding free spaces.

Taking into account the fact that the young generation is typically small, large objects would fill it at a high rate and would disrupt the collection frequency. For this reason, these objects, which exceed the default size of a TLAB, are allocated directly in the old generation. As mentioned in Section 2.2.4 objects are usually small, thus the number of large objects allocated tend to be small in most applications. Therefore the allocation in the old generation is done directly from the heap without making use of a TLAB.

4.4 Collect Heap Operation

The CollectHeap class is a subclass of GCOperation, the type which the VMOperationThread (GC Thread) expects as a parameter to run the garbage collection routine. In addition the CollectHeap class is contained in the body of the GenerationalHeapScheme class, as a final one. The reason behind this is the fact that it uses a large amount of functionality contained in its outer class.

This operation performs the actual garbage collection on the generational heap. It uses cell visitor agents to access the objects stored in the heap and the latter further deploys reference updating agents to copy the referenced objects depending on what part of the heap is being collected. In addition, two other classes are used in order to specify the policy relating to special reference handling .

4.4.1 Implemented Generational Garbage Collection

Garbage collections are scheduled whenever the part of the heap in which allocation takes place becomes full. At that point a mechanism presented in Section 3.3 stops the mutator threads and starts the garbage collector thread to perform the memory reclamation routine. There are two types of reclamation routines which are used according to what part of memory has been exhausted: minor collections and full collections.

4.4.2 Minor Collection

A minor collection is the process of reclaiming the memory associated with dead objects from the young generation. It is triggered when the young allocation space becomes full. Firstly, the detection of live objects is performed, with the result of copying all found structures to the old generation. At the end of this process, the entirety of the young generation's memory is reclaimed and used for further allocation. Objects stored in the old generation are assumed to be alive in further minor collections.

4.4.3 Full Collection

A full collection is performed when the memory associated with the old generation is exhausted. A semispace garbage collection, reused from the Maxine VM project, is performed on the filled heap partition to reclaim unused memory.

4.4.4 Live Object Discovery

Objects are classified to be alive by following a path of pointers starting from the root set. The Maxine VM, in addition to the mutator roots present in the thread stacks, local variables and registers, has a number of other objects that are considered program roots: the boot heap, code cache and the immortal heap. For the purpose of the young space collection, another set of objects must be added to the root set, namely the objects which reside in the old generation. Due to its significant size, it would be expensive to follow the pointer trail starting from all old objects. Furthermore, intergenerational pointers are not numerous, thus the object tracing would be wasteful with regard to performance if all promoted objects would be treated as roots. In consequence, a card table is used to check which objects may hold references to eden.

These root sets have been developed using the visitor pattern. They all provide the functionality of iterating through their referenced objects and the visiting classes may perform their desired actions on these objects (see Figure 4.3.1). Figure 4.4 illustrates the discovery of live objects during garbage collection. The CollectHeap entity scans the root sets for reachable objects by passing the visitors, represented as circles (excluding the circle labeled as CollectHeap), to the required root sets, as parameters. The colour is consistent with the set the agent visits.

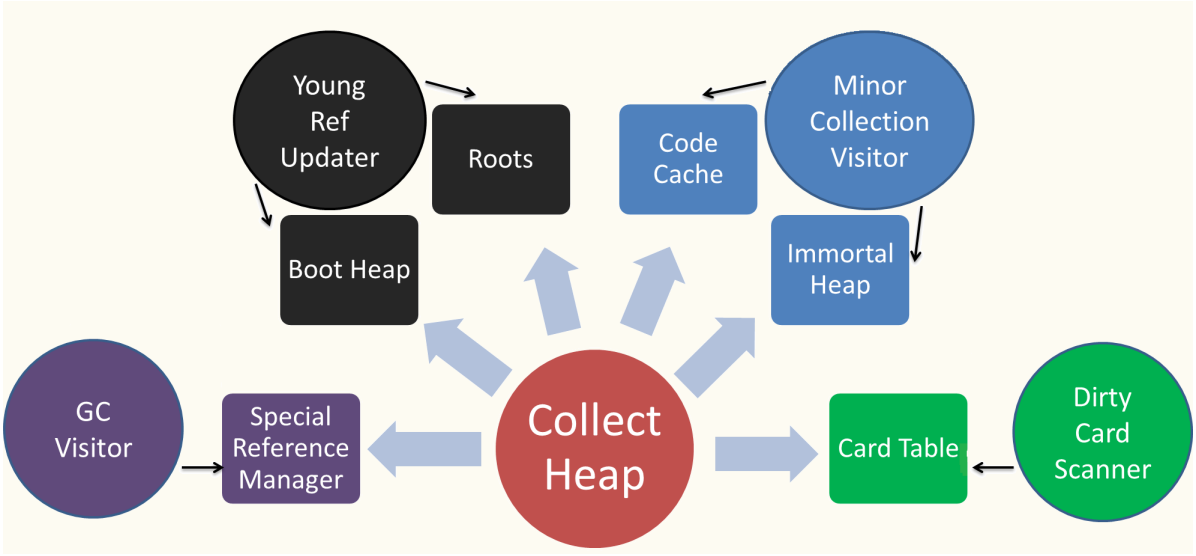


Figure 4.4: Live object discovery in the Maxine VM.

4.5 Visitor agents

The responsibility of accessing objects during the liveness analysis is given to the MinorCollectionVisitor and FullCollectionVisitor classes, corresponding to the generation currently being collected. They implement the CellVisitor interface to be accepted as visitors by the root sets. They access the current object and pass the responsibility of copying the referenced structures to the ReferenceUpdater visitors.

The ReferenceMapper abstract class contains the functionality of copying the actual objects between partitions. It also implements the PointerIndexVisitor interface to be recognised as a visiting class. The YoungRefUpdater and OldGenRefUpdater inherit the copying facilities and the title of visitors from the ReferenceMapper. They copy the objects referenced by the current object and update its fields with the new locations.

The last visitor type classes presented are used to specify the policy regarding the special references. The GC and FullGC classes implement the SpecialReferenceManager.GC interface, used by the SpecialReferenceManager class to perform the action which relates to object preservation. These two agents also provide information about how the reachability of an object is to be determined by the manager. In addition, these two visitors are implemented as final classes within the GenerationalHeapScheme for the same reason as the CollectHeap.

Chapter 5

Implementation

The description of the development process used during the project is given, followed by the key implementation details of the system presented in the previous chapter to reflect the implementation stage of this project. In addition, it is worth noting that all algorithms presented in this chapter are illustrated using pseudocode.

5.1 Development process

The development of the garbage collector has been done using an incremental approach. This was appropriate because although the aim of the project had been to learn about garbage collection internals, emphasis has been put on developing a robust piece of software. In consequence, the functionality has been incrementally implemented and tested.

The first feature to be implemented was the heap initialization and object allocation, followed by the large object policy tested before the iteration to a minor collection routine. The collection routine was at first developed by naively scanning all the objects allocated in the old generation, effectively considering them all to be roots, during live object analysis, rather than making use of a remembered set. In addition, the card table and write barrier have not been made available to the Maxine VM community at the start of the project, thus there was a need to synchronize the iteration of upgrading the minor collection routines with the release of the card table by the Maxine team in December 2011. There has been continuous work done to improve and fix bugs after its release as well.

5.2 System metrics

The application code used to implement the system consists of:

- 11 classes (of which 3 are inner classes of the GenerationalHeapScheme)
- 1 interface
- 2350 lines of code

These classes interact with others implemented previously as part of the Maxine VM project which in turn interact with many more. There are a number of classes which needed to be understood and used to deliver the generational garbage collector. Not all were particularly

well documented and their functionality was derived from examples of their use as well as from their code. The numbers are:

- 30 classes
- 5 interfaces

5.3 Generational Heap Scheme

As described in Section 4.3, the logic of initializing the heap and object allocation is factored in the GenerationalHeapScheme class. This section will explore their implementation details.

5.3.1 Heap initialization

As specified in Section 4.3.1 the memory regions in the heap must be aligned. A large amount of virtual memory is reserved during boot image generation to avoid the allocation of the memory resources to other applications during the allocation of the memory structures belonging to the VM. The desired amount is passed by the heap scheme to the Maxine's boot image generator to reserve the actual virtual memory and to map the boot heap, followed by the code cache at the start of the new area. At this point allocation of the generations is done at the end of the code region. The pseudocode of this mechanism is shown in Algorithm 5.1.

```
method: allocateSpaceAt(address, space, size)

begin

  VirtualMemory.commitMemory(address, size);
  set space start adress = adress;
  set space size = size;
  set allocation mark = adress;

end

release space:
VirtualMemory.uncommit(endOfCardTableRegion, unusedSpaceSize);
```

Algorithm 5.1: Allocation of generations and the deallocation of unused space.

The VirtualMemory class is used to allocate a generation at a particular adress. In addition the memory used to store the card table is also allocated, at the end location of the old generation, specifically the old generation-to space. The last step of initialization is to release the unused reserved space to the operating system.

5.3.2 Allocation

Objects are, by default, allocated in the young generation using TLABs. Effectively, most requests for allocation are ones demanded for these buffers.(see Section 4.3.2). The allocation from the heap is done by incrementing a pointer which represents the current allocation mark of eden with the size of the resource and returning the start address of the newly reserved cell.

Requests for object allocation are performed by multiple threads, thus updates of the allocation mark are achieved using compare and swap. This thread synchronization technique yields lower overhead than using a Java ”synchronized” statement. The functionality of the former is to atomically test and modify a shared resource by comparing its contents with a given value, in this case the expected value of the allocation mark, and performing the write of the new value if and only if the previous test has been successful.

Large object allocation requests are satisfied with the memory resources of the old generation to space by using the same mechanism as the allocation of TLABs. However, the use of the old generation comes with the risk of exhausting the memory associated with it. Therefore, additional checks are required to avoid this. At each allocation in the old generation, the current allocation mark is compared with a predefined threshold, and if the mark exceeds it, a full collection is scheduled. In addition, the card table maintains a record of the address identifying the first object which overlaps with a card. In consequence after the chunk of memory from the old generation has been reserved, the card table must be informed of the start address and the size of the object subject to allocation, to be able to update its first object table.

5.4 Collect Heap Operation

The CollectHeap class contains the methods used to collect the generations. This section further presents the implementation details relating to this functionality.

```
method: void collect()
begin:

  if(full collection triggered)
    perform fullCollection and exit;

  scanRoots(refUpdater);
  scanBootHeap(refUpdater);
  scanCode(minorCollectionVisitor);
  scanImmortalHeap(minorCollectionVisitor);
  scanCardTable(dirtyCardScanner);
  moveReachableObjects(allocationMarkAfterLastCollection);

  allocMarkBeforeSpecialRefProc = getOldGenerationToMark();

  SpecialReferenceManager.process(GC);
  moveReachableObjects(allocMarkBeforeSpecialRefProc);

  allocationMarkAfterLastCollection = getOldGenerationToMark();

  if(full collection triggered)
    perform full collection;
end
```

Algorithm 5.2: Minor collection routine.

5.4.1 Minor Collections

Algorithm 5.2 shows the pseudocode for performing a minor collection. The routine commences with a check for the need of performing a full collection which may have been requested by a large object allocation. If so, the current collection is suspended and a routine to collect the old generation is performed. Otherwise, the minor collection proceeds. During each scan of starting sets of objects, reachable objects are copied to the old generation.

The updating of references in object fields in the young generation to other objects that have been copied is done by a visitor class, the YoungRefUpdater. This agent is passed as a parameter to the predefined method of scanning all references in the boot heap and in the mutator roots.

The immortal heap and code cache do not have reference maps associated, thus the whole memory space must be iterated through and each object must be accessed for information about other reference types it refers to. The MinorCollectionVisitor agent has been developed to meet this requirement.

The next step in the collection routine is to scan the card table. A visitor object is passed to its scan method to visit each cell in the range derived from the dirty cards.

At this point all the root sets have been scanned and all objects directly referenced by these roots have been copied to the old generation. However, the remaining objects are not. Figure 5.1 illustrates the old generation to space in the process of copying the other reachable objects to the old generation. Orange cells represent the objects allocated in the old generation prior to the current collection; brown cells represent objects copied in the current minor reclamation routine; purple cells are the rest of the reachable objects in the process of being copied; the stripe pattern on a cell shows the object whose fields are currently checked for references to others; the first, second and final step of the process is shown. A pointer of the last allocation in the old generation to space is kept between minor collections; this pointer shows the exact location of where the first object was copied during the current collection. Therefore, by visiting each of these objects, using the MinorCollection agent, and examining their reference fields, until the current allocation mark of the old generation has been reached, all live objects can be discovered and promoted. This technique is effectively an implementation of the graph reachability algorithm.

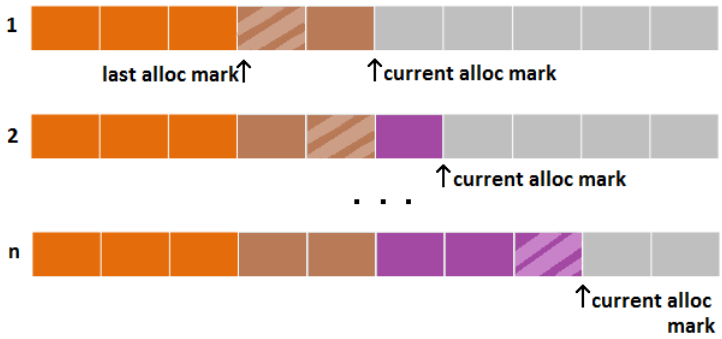


Figure 5.1: Old generation to space during the move reachable phase.

The next step of the minor collection routine is the processing of special references. This procedure is factored in the SpecialReferenceManager class. For this processing it requires knowledge about how reachability is defined in this particular heap scheme and the action that must be performed in order to preserve the object. These specifications were developed in the

GC final class within the GenerationalHeapScheme. The former implements the SpecialReferenceManager.GC interface. The GC class is passed to the processing method of the SpecialReferenceManager as a parameter. Processed special referenced objects should be scanned to discover objects reachable through them, resulting in another graph reachability routine to be performed.

Before the garbage collection finishes, it checks if a full collection might have been triggered by copying the objects to the old generation to space. If so, a full collection is started, otherwise the VMOperationThread gives up control to the mutator threads and the running of the application may resume.

5.4.2 Full collection

The full collection has the same phases of live object discovery as the minor collection, with the exception of not using a card table. Instead, while moving the rest of the reachable objects, it starts scanning the old generation to space (recall Section 2.2.3) from its first object allocated, rather than using the last known allocation mark (see Figure 5.1). Discovered live objects are copied to the old generation to space and after this is done, the mutator threads may be restarted. As in the previous collections, agents are implemented to be passed to the root sets scans.

5.5 Reference Mapper

The ReferenceMapper class contains the functionality of copying objects between heap partitions. It is declared as abstract, thus the actual copying action is performed by its subclasses. These methods require a reference as a parameter and copy the object associated with it in the required heap partition. The pseudocode of the method used to copy an object from eden to the old generation to space is shown to facilitate the reader's understanding.

```
method: Reference mapRefFromEden(Reference oldReference)

begin:

  origin = getObjectStartAdress();
  if (oldGenerationTo.contains(origin)
    return oldReference;
  read forward reference;
  if(forward reference != 0)
    return forward reference;
  get object size;
  allocate resource from the Old Generation-To space;
  copyBytes from eden cell to old generation cell;
  write new forward reference;
  return new reference;

end
```

Algorithm 5.3: Method used to copy objects during a minor collection.

Firstly, the object's origin address is derived from the reference and it is checked if it resides in the old generation. Old objects do not require copying, therefore their current reference is returned. If its header contains a forward reference, its value is returned, as the object has been copied previously. Otherwise, a specified chunk of memory is allocated from the old generation, and the bytes of the object are copied to the newly allocated cell. Lastly, a forwarding reference to the old generation cell is installed in the object's header located at the old address before the reference to the new location is returned.

Additional operations need to be performed during the allocation of the old space resources, because they can be exhausted. Therefore, before releasing a chunk of memory for allocation, it is checked that the current allocation mark has not exceeded a threshold. If it has, a full collection is scheduled. The threshold gives headroom for the minor collection to finish before running the reclamation routine on the old generation. In addition after the space has been reserved, the card table must be informed of the allocation to update its first object table.

5.6 Visitor agents

The MinorCollectionVisitor, FullCollectionVisitor and the DirtyCardScanner have similar functions. Their visit() method accesses the hub pointed to by the current object and retrieves information about its memory representation: tuple, array or hybrid (See Section 3.4)). According to the type of representation, a reference updating agent may be passed as a parameter to visit the tuple's reference map, in the case of tuple and hybrid layouts, or a separate mechanism, factored in the ReferenceMapper class, is used in the case of array layouts. The latter, reads the array length from the header of the current object and iterates through the fields copying objects and updating the references to them. In addition, as described in Section 3.5 special references may be discovered in the fields belonging to the current object; these are appended to a "discovery list" managed by the SpecialReferenceManager to be processed later.

The reference updating agents are the YoungRefUpdater and OldGenRefUpdater classes. Their visit method access accesses a reference at a specified index withing the body (field) of an object, copies the object it refers to, and set the value of the current field to the reference pointing to the new location.

GC and FullGC are the last visitor agents implemented by this project. As stated in Section 5.4.1, their responsibility is to define reachability and the policy of preserving live objects in the current heap scheme.

During minor collection, reachability is defined by the GC class as objects residing in the old generation to space or located in eden with an existent forward reference installed in their header. On the other hand, object reachability during the collection of the old generation is specified by the FullGC class as objects with existing forward references in their headers, or objects residing in the old generation to space. Recall that the two semispaces are swapped at the start of a semispace collection, therefore the old generation to space is the partition in which live objects will be copied.

The second information needed by the SpecialReferenceManager is how live objects are handled by the garbage collector. Therefore, the preserve method called on these agents should be either a promotion to the old generation to space, or the copying of the specified object to the old generation to space.

5.7 Challenges

Several difficulties have been encountered throughout the implementation of this project. One of the most challenging aspects was the fact that the implementation shifted away from the average programming paradigm with regards to debugging. Stack traces exist to provide a relative position in which the virtual machine has crashed, but they were not always helpful. For example, if the garbage collector would fail to detect a live object, the halt of the VM would have happened during the mutating phase rather than in the garbage collection one. Therefore it has been difficult to work out which part of the garbage collector did not fulfill its functionality. Several other errors may appear in this particular situation, for example the stack might fill up due to the amount of exceptions which may have been thrown giving the simple message of ”stack overflow” without any knowledge of the reason it occurred.

Another challenging aspect was the initialization of the card table. Although the final artefact does not exhibit the complexity of this action, there have been several “trial and error” runs to provide correct behaviour. The reason for this was the lack of documentation of this new facility.

Furthermore, the Maxine VM project is under continuous development, therefore this project had to be kept up to date with the latest changes in order to take advantage of them.

Chapter 6

Testing and Evaluation

During the development of the heap scheme required to implement the generational garbage collector several methods of testing have been used. In addition experiments have been performed to understand the garbage collector's configuration for performance and to compare the Maxine VM's semispace collector (the existing garbage collector in the VM) with the one implemented in this project. Recall that this implementation of the generational garbage collector directly copies live objects found during a minor collection to the old generation, without having any aging mechanism.

6.1 Test cases

The virtual machine is a very complex piece of software (as indicated in Chapter 3) and it needs most of its components to work in order to run a program. Thus it is not possible to run single components and test them individually. The only way a garbage collector can be tested is by running certain programs on the virtual machine and, possibly, generating useful output during the runtime of the application. Figure 6.1 shows an example of a test case: a small program used to allocate large objects and the output generated by the virtual machine. This information provides evidence that large objects were correctly allocated in the old generation.

6.2 Maxine tests

The garbage collected heap scheme was also tested using the unofficial tests from the Maxine VM project. These tests are designed to test key features of the virtual machine including garbage collection. More specific, there are 10 garbage collection test cases implemented in the Maxine Project. The complexity of the programs range from as simple as a call to System.gc() to allocation of large binary trees and large lists within lists. A large number of collections may be forced by these tests, as shown in Figure 6.2 in which there are 40 minor collections and 4 full collections. Figure 6.3 illustrates output of another test program which allocates a list of long lived binary trees along with similar structures that become garbage immediately after. The details of objects promotion is contained in the output generated by the running VM; it can be seen that live objects are correctly copied to the old generation.

LargeObjects.java

```java
import java.util.*;
public class LargeObjects
{
  public static void main(String[] args)
  {
    int i = 0;
    int noOfIterations = 100;
    while(i < noOfIterations)
    {
      //Create large object
      int[] largeArray = new int[1000000];
      Arrays.fill(largeArray,0);
      i++;
    }//while
  }//main
}
```

```
End: 0x7f4ec228d000
****** Old Generation From Space ******
Start: 0x7f4ec228d000
End: 0x7f4edbc0d000
****** Old Generation To Space ******
Start: 0x7f4edbc0d000
End: 0x7f4ef558d000
****** Card Table ******
Start: 0x7f4ef55c3000
End: 0x7f4ef5a8d030
****** End of reserved space ******
0x7f5028bfa000
allocating large obj to 0x7f4edbc0d000
allocating large obj to 0x7f4edbfdd918
allocating large obj to 0x7f4edc3ae230
allocating large obj to 0x7f4edc77eb48
allocating large obj to 0x7f4edcb4f460
allocating large obj to 0x7f4edcf1fd78
allocating large obj to 0x7f4edd2f0690
allocating large obj to 0x7f4edd6c0fa8
allocating large obj to 0x7f4edda918c0
allocating large obj to 0x7f4edde621d8
allocating large obj to 0x7f4ede232af0
allocating large obj to 0x7f4ede603408
```

Figure 6.1: Testing Large Object Allocation.

```
--Start GC 43--
--Before GC   used: 122787 Kb, free: 28 Kb --
Minor Collection started
Timings (ms) for GC 43: clear & initialize=0, root scan=0, boot heap scan=73, co
de scan=0, immortal heap scan=0, card table scan=5, copy=0, weak refs=0, total=7
8
total:
3483
Minor collection finished
--After GC    used: 0 Kb, free: 122816 Kb, reclaimed: 122787 Kb --
--End GC 43--
Stack reference map preparation time: 0ms
--GC requested by thread main[id=1] freed enough--
GCTest5 done.
Timings (ms) for all GC: clear & initialize=0, root scan=1, boot heap scan=3197,
 code scan=0, card table scan=256, copy=26, weak refs=2, total=3482
Number of un countered full collections: 0
Number of mutator triggered full collections: 4
andr3iro90@mcore:/scratch/andr3iro90/Project/ThirdYearProjectFinal/maxineFinal>
```

Figure 6.2: GCTest5 with a high number of collections.

6.3 Compliance Tests

The newly modified virtual machine was tested on the DaCapo benchmarks [18]. These benchmarks are real world, open source programs with non-trivial memory load, carefully selected to test the key components of a JVM. The two DaCapo releases have been the result of a total of 8 years of development. The version of the Maxine VM modified in this project, successfully passed the following benchmarks: sunflow, avrora, lusearch and luindex. One of the reasons for the rest not working could be the fact that the card table is currently still developed and there might still be a number of bugs which have not been discovered. Figure 6.4 illustrates the VM successfully running the benchmarks with a heap size of 2 GB of which the young generation represents 20% and the default TLAB size is 64 KB.

```
****** Eden ******
Start: 0x7fc63d641000
End: 0x7fc64a341000
****** Old Generation From Space ******
Start: 0x7fc64a341000
End: 0x7fc663cc1000
****** Old Generation To Space ******
Start: 0x7fc663cc1000
End: 0x7fc67d641000
```

```
BEGIN: Moving reachable
Visiting cell 0x7fc6668a3bf8
Eden->old
Forwarding java.io.FileDescriptor from 0x7fc63d676568 to 0x7fc6668a3c58 [24 byte
s]
Visiting cell 0x7fc6668a3c18
Eden->old
Forwarding java.io.FileDescriptor from 0x7fc63d71ba20 to 0x7fc6668a3c70 [24 byte
s]
Visiting cell 0x7fc6668a3c38
Eden->old
Forwarding java.io.FileDescriptor from 0x7fc63d7394c0 to 0x7fc6668a3c88 [24 byte
s]
Visiting cell 0x7fc6668a3c58
Visiting cell 0x7fc6668a3c70
Visiting cell 0x7fc6668a3c88
END: Moving reachable
```

Figure 6.3: GCTest3 and promotion details.

```
voicula9@voicula9-Studio-1557:~/maxineFinal$ mx vm -Xmx2g -XX
:YoungPercent=20 -jar dacapo-9.12-bach.jar avrora luindex lus
earch sunflow > benchmarkScrapInfo
===== DaCapo 9.12 avrora starting =====
===== DaCapo 9.12 avrora PASSED in 15470 msec =====
===== DaCapo 9.12 luindex starting =====
===== DaCapo 9.12 luindex PASSED in 2361 msec =====
===== DaCapo 9.12 lusearch starting =====
===== DaCapo 9.12 lusearch PASSED in 7824 msec =====
===== DaCapo 9.12 sunflow starting =====
===== DaCapo 9.12 sunflow PASSED in 11326 msec =====
```

Figure 6.4: DaCapo benchmaks on the Maxine VM.

6.4 Experiments

An important aspect concerning the generational garbage collector is the fact that it is highly tunable. The size of the heap and the ratio of the young generation to the old generation may influence the performance of the collector. These values can be specified when starting the virtual machine from the command line. The size of the heap can be specified using the option "-Xmx". Furthermore, the percentage of the total heap size reserved for the young generation can also be set using "-XX:YoungPercent=".

The remaining of this section shows various experiments to support this claim. All measurements have been made on a machine with an Intel Core I7 720 QM processor running at 1.60 Ghz frequency (2.8GHz maximum frequency) with 4GB of main memory under the Ubuntu 11.10 64 bit operating system. The Maxine VM boot image was generated for this platform using the JDK version 1.7.0_147-icedtea. All measurement have been repeated 10 times using the sunflow benchmark of the DaCapo 9.12 suite and the reported values represent the average of these measurements. This benchmark was chosen to perform the experiments

with because it requires several garbage collections to be called during its run. Its functionality is to render a set of images using ray tracing [19].

6.4.1 Heap Size

The first experiment presented is the effect of varying the heap size on the application's total run time and garbage collection time. Figure 6.5 shows the results of this experiment using the young generation as 20% of the total heap size and the default TLAB size. As a convention, in the following plots black is intended to represent the application runtime, while red represents the garbage collection time, unless stated otherwise.

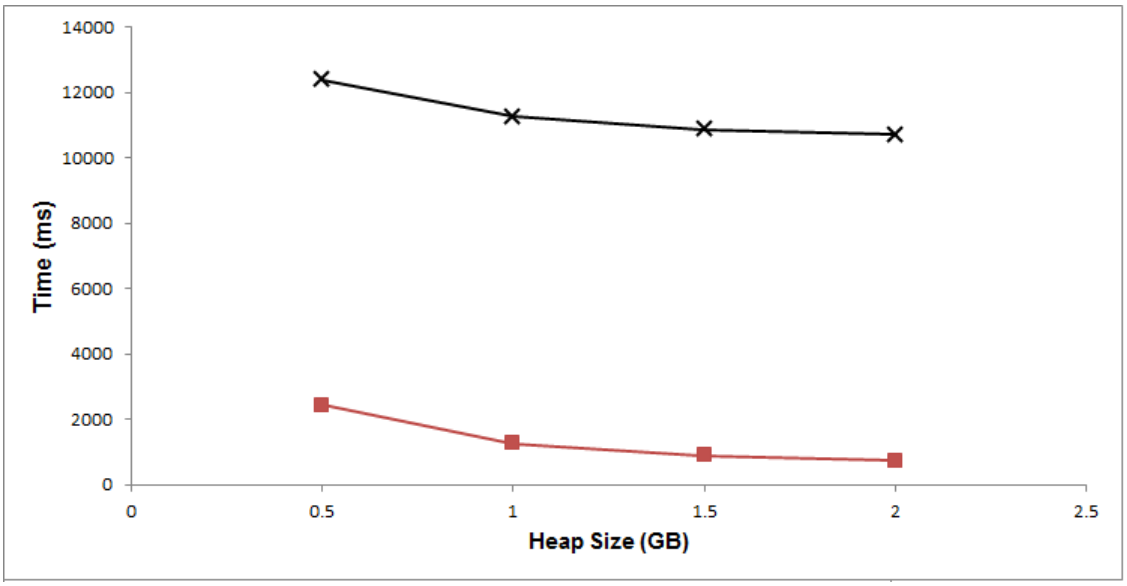


Figure 6.5: Application run time (black) and garbage collection time (red) as a function of the heap size.

Heap Size (GB)	Application's run time STD	GC Time STD
0.5	46.5944203	10.8853112
1	62.9111278	16.3122653
1.5	37.9168828	10.68644
2	48.7221715	17.4802746

Table 6.1: Standard deviation of the measurements with different heap sizes.

Table 6.1 contains the standard deviations of the measurements performed. Lower garbage collection times imply better performance. The plot in Figure 6.5 shows how the application's run time decreases while the heap size is increased. The rate of the performance increase is proportional with the percentage increase of the heap size. In addition, the garbage collection time's rate of change is similar to the application's run time, thus it is obvious that the program's performance is directly influenced by the garbage collector.

This experiment's result that garbage collection time decreases with larger heaps was expected for the reason that a larger heap would fill up at a lower rate than a smaller one would. This implies that garbage collections will not be required as often. The collection of a larger

heap would require more time to complete, namely the time spent in any single garbage collection should be higher than the one using a smaller heap. However, a second consequence of having a larger heap is the fact that keeping objects in the young generation for a longer time gives them a higher chance of death, thus the number of live objects will not usually be considerably higher; this implies good scalability. As the generational garbage collector copies live objects, its time complexity will be proportional to the number of live objects. Therefore, an increase of the heap's size will not necessarily imply an identical proportion of increase in the individual garbage collection times. Table 6.2 shows the total time spent in garbage collection and the time of an individual collection (pause time) using different heap sizes. The reason for the increase in the pause times in the larger heaps is because the number of live objects to be copied to the old generation to space is increased.

Heap Size (GB)	GC Time (ms)	No. of GCs	Individual GC Time (ms)
0.5	2425.9	26	93.30
1	1270.9	13	97.76
1.5	905	9	100.55
2	728.2	7	102.86

Table 6.2: Measurements of the garbage collection times for different heap sizes.

Although the heap size increases by 100%, 50%, 25% and the total garbage collection time decreases by 50%, 29%, 19%, the individual GC time only increases by 4.7%, 2.86% and respectively 2.29%. In conclusion, this experiment shows that using this benchmark, having a larger heap with a proportionally increased eden implies a lower total time spent doing garbage collection, but will slightly increase the time spent in each garbage collection cycle.

6.4.2 Young Generation Size

The next experiment has been performed to determine what particular percentage of the total heap size should be reserved for the young generation in order to achieve performance. The measurements have been produced using a heap size of 2 GB and the default TLAB size. Figure 6.6 shows the result of this experiment; the X-axis represents the percentage of the heap size allocated for the young generation, while the Y-axis represents the time in milliseconds. Table 6.3 contains the standard deviations of all measurements performed for this experiment.

Young Space (%)	Application's run time STD	GC Time STD
5	58.277	10.04
10	93.23	10.92
15	56.12	24.93
20	29.1419	9.88
25	60.89	16.86
33	70.89	21.61

Table 6.3: Standard deviation of the measurements performed on different young generation sizes as a percentage of the heap's size.

Figure 6.6 shows that the application's run time and garbage collection time decrease proportionally with the increase of the young generation's percentage of allocated heap space. A

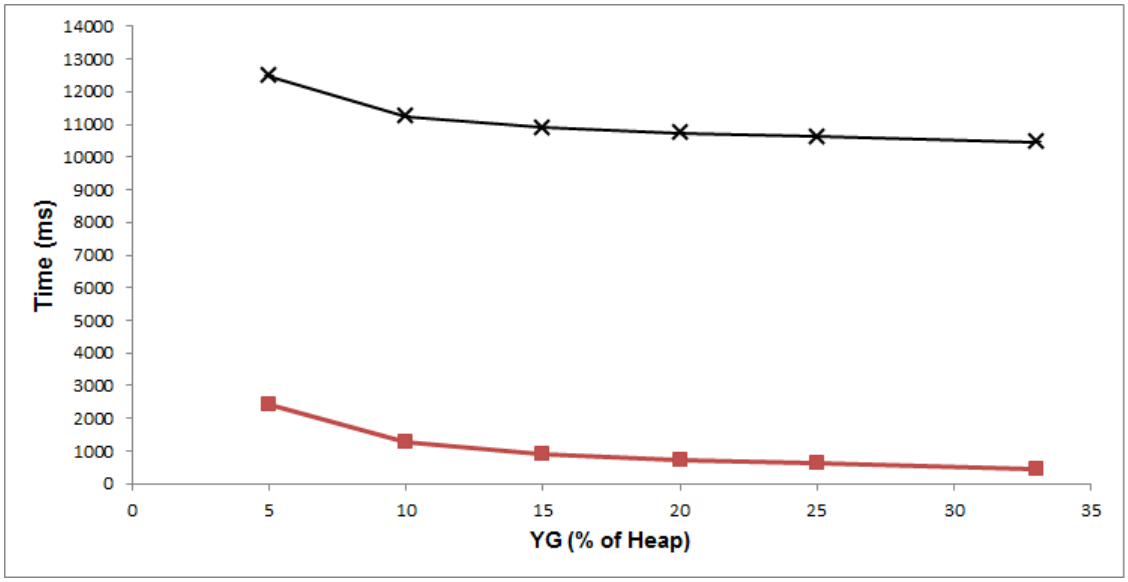


Figure 6.6: Application's run time (black) and garbage collection time (red) as a function of the percentage of heap size reserved for the young generation.

small allocation space implies a fast garbage collection cycle, but very frequent collections; the total time spent doing garbage collection is expected to be high. This claim is supported by the experiment. It is illustrated in the plot that the application's run time and garbage collection time is much higher at smaller percentages. Also Figure 6.7 shows how the average value of a single collection evolves with regard to these young generation percentages. Judging from the latter, it is clear that the previous claim of individual garbage collections being faster for a smaller allocation space is true. A change in the rate of the increase in the time spent in individual collections can be observed. The reason for this may be the fact that a large number of objects might have died just after the 15 % boundary, or perhaps the objects allocated in the 5 % added to the young generation have died at a higher rate.

Figure 6.8 illustrates how the occupancy of the old generation evolves while varying the young generation's size. The X-axis represents percentages of the heap size reserved for eden and the Y-axis contains the space occupied in the old generation to space. The standard deviations of the measurements of old generation occupation are contained in Table 6.4. This result is consistent with the claim that a large allocation space would allow objects more time to die before being subject to garbage collection than a smaller space would. This choice would, therefore, reduce the time for full collections. In addition, the change between the occupancy at the 15% and 20% boundaries is not as significant as between the other sizes. This suggests that these two young space sizes are similar with regard to dead object filtering. However, according to the plot, the highest filtering of dead objects is achieved with a 33% of the heap space allocated for the young generation, in the context of this benchmark.

Another fact which must be taken into account is the effect of full collections being scheduled when the old generation to space is exhausted. Due to the fact that the benchmarks successfully ran on the modified Maxine VM do not fill up the old generation to space (tested with a 512m and a 5% space allocated for the young generation) the experiment will be done on an unofficial test from the Maxine VM project: GCTest8. This is a heavily multi threaded test in which each thread creates various kinds of garbage. The experiment was performed using a heap size of 1 GB with variable percentages of the heap's size reserved for eden.

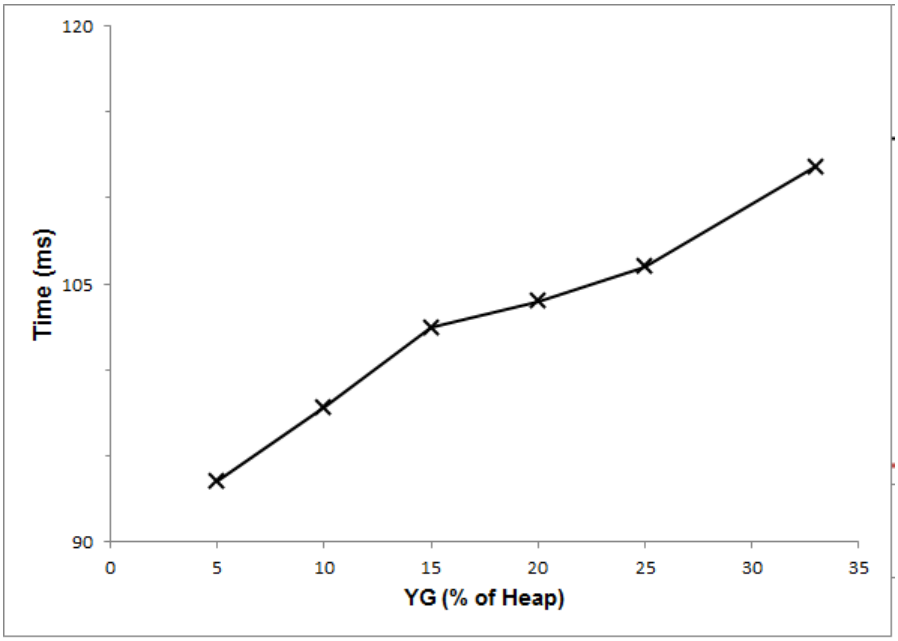


Figure 6.7: Average individual garbage collection times as a function of the young generation's percentage of the heap's size (2 GB).

Young Space (%)	Occupancy STD
5	0.422
10	0.44
15	0.23
20	0.22
25	0.17
33	0.018

Table 6.4: Standard deviations of the measurements for old generation occupancy.

Table 6.5 contains the results of running the experiment. The full collections have all been triggered by a minor collection, thus the cost of that particular cycle would be the addition of the minor collection time plus the full collection time, which is more than twice the average GC individual time. The number of full collections vary according to the size of the young generation. As the allocation space increases in size, the number of full collections decreases. The larger space filters the dead objects better than smaller ones, thus the number of full collections is indeed decreased. From these experiments it can be deduced that, in the context of this

Young Size(%)	GC#	Full Coll#.	Indiv. GC Time(ms)	Full Coll. Time(ms)
6	130	3	94.39	187
7	111	2	95.05	192
15	52	1	98.17	196
33	23	1	103.6	196

Table 6.5: GCTest8 garbage collection information.

benchmark, the choice of the size chosen for the young generation can have a big impact on the

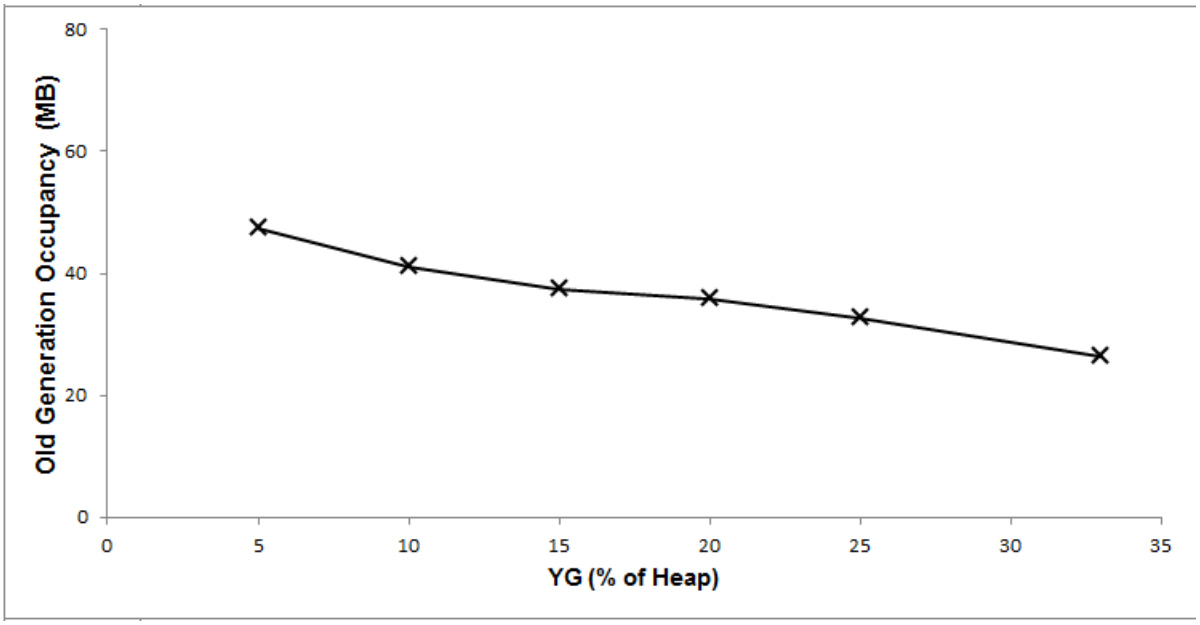


Figure 6.8: Old generation occupancy plotted against the young generation as a percentage of the heap size.

run time of the application. Although larger young generations slow down the individual collection cycles, overall it would provide fewer full collections and better application throughput (see Section 2.1).

6.4.3 Generational Garbage Collector vs Semispace Garbage Collector

The last experiment presented in this chapter is the comparison between the Maxine VM's semispace garbage collector and the one implemented in this project, the generational garbage collector. Based on the previous results, performance with regard to application throughput for the generational collector is achieved by having a large percentage of the heap space allocated to the young generation. Therefore, the experiment has been performed using a 33% young space percentage and the default TLAB size. Figure 6.9 and Figure 6.10 show the application's run time and, respectively garbage collection time plotted against the size of the heap by using both garbage collectors. Blue represents the benchmark ran using the generational garbage collector, while red represents the one using the semispace collector. Figure 6.11 shows the average time of a garbage collection cycle as a function of the heap size for both collectors. The X-axis represents the heap's size and the Y-axis represents the time. The standard deviations for the measurements used in this experiment are contained in Table 6.8 and Table 6.9.

Heap Size (GB)	Performance Increase - Total GC Time(%)
0.5	15.7
1	7.9
1.5	21.2
2	12.2

Table 6.6: Performance measures of the semispace collector compared to the generational collector.

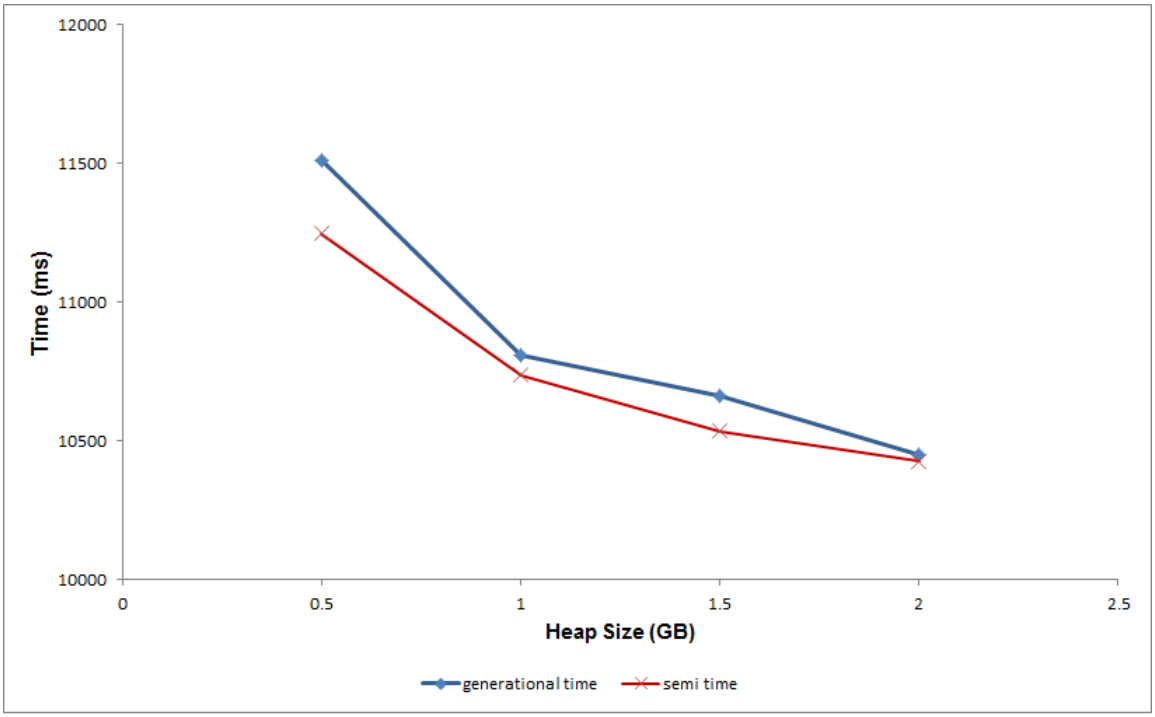


Figure 6.9: Application's run time against heap size using the generational (blue) and semispace collector (red).

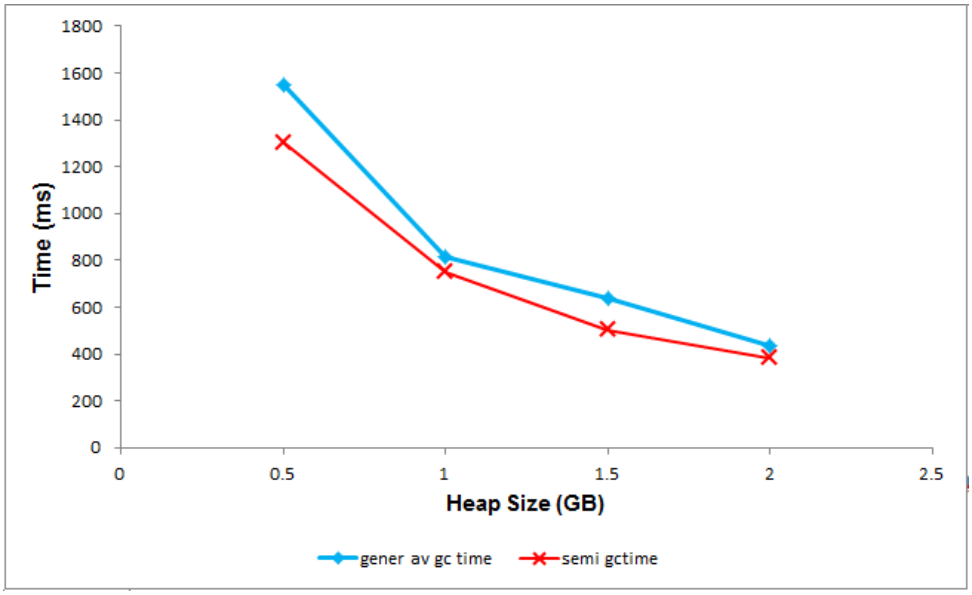


Figure 6.10: Total garbage collection time against heap size using the generational (blue) and semispace collector (red).

It can be observed that the semispace collector has a better performance in terms of total garbage collection time. The reason why the semispace collector yields better throughput, can be the fact that garbage collections are less frequent than in its counter part. Table 6.6 contains specific information about the semispace collector's performance increase compared to the generational one.

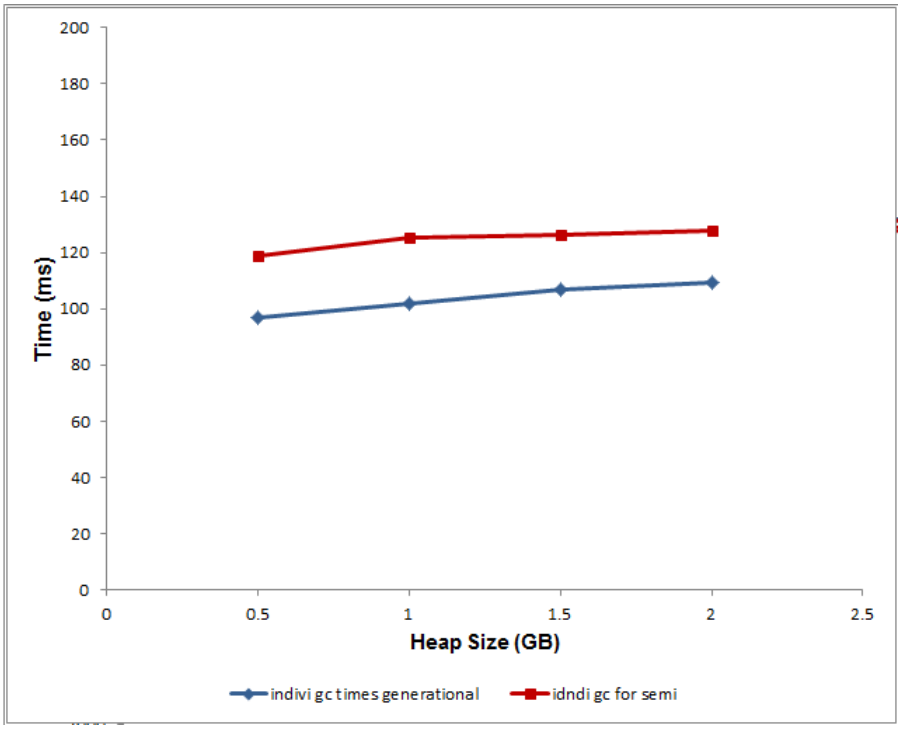


Figure 6.11: Average time of a garbage collection cycle against heap size using the generational (blue) and the semispace collector (red).

Heap Size (GB)	Performance Increase - Average Pause Times(%)
0.5	18.4
1	18.5
1.5	15.32
2	14.5

Table 6.7: Performance measures of the generational compared to the semispace collector.

However, the generational garbage collector has lower pause times compared to the semispace one. A minor collection reclaims a smaller heap, thus its collections would be expected to be less time consuming. Another reason might be the fact that it does not repeatedly copy long lived data, therefore achieving higher efficiency. The percentage increase of the collector's performance in terms of pause times is contained in Table 6.7

Heap Size (GB)	Application's run time STD	GC Time STD
0.5	51.56	17.67
1	77.86	9.904
1.5	44.65	15.72
2	57.39	11.01

Table 6.8: Standard deviation of the measurements using the generational garbage collector.

In conclusion, this experiment shows the semispace collector exhibiting better throughput than the generational collector while the latter dominates in terms of average time spent in a single garbage collection cycle.

Heap Size (GB)	Application's run time STD	GC Time STD
0.5	42.96	8.11
1	86.58	20.37
1.5	52.37	9.96
2	58.14	6.93

Table 6.9: Standard deviation of the measurements using the semispace garbage collector.

6.5 Summary

In summary, this chapter presented the ways in which the generational garbage collector has been tested. The experiments have shown that, in the context of the sunflow benchmark, performance is achieved by using a large heap of which 33% should be used for allocation. In addition, the comparison between the semispace and generational garbage collectors came to the conclusion that the former achieves better performance in terms of throughput, while the latter induces lower pause times in the same context.

Chapter 7

Conclusion

The project presented in the report has been successful; a working generational garbage collector has been sucessfully delivered. The collection of the old generation is achieved by performing a semispace collection routine reused from the Maxine VM project. In addition, no aging criteria is used by this implementation, therefore all live objects are copied to the old generation as they are discovered. The various experiments have proposed a configuration for the generational garbage collector and have shown good performance with regard to pause times compared to the garbage collector in use by the Maxine VM while running the sunflow DaCapo benchmark.

7.1 Future Work

This implementation of the generational garbage collector can be upgraded to a parallel version. An example of work which could be parallelised is the copying of objects. This could be done using TLABs to avoid thread races and the object header could be used to set a "GC" bit to let the other threads know it is being copied.

In addition, concurrent versions of the generational garbage collectors exist, which can be studied using this implementation as base functionality. Different ways of collecting the old generations may also be explored to check their impact on performance.

By having understood garbage collection internals, other interesting areas of this field can be explored. Research can be undertaken to develop new techniques for garbage collection, such as having it cache aware to make the behaviour of the collector more predictable.

References

[1] "Threads - Maxine VM - Oracle Wiki." https://wikis.oracle.com/display/MaxineVM/Threads, last visited 16th April 2012.

[2] "Objects - Maxine VM - Oracle Wiki." https://wikis.oracle.com/display/MaxineVM/Objects#Objects-TheMaxineProject\%3AObjectrepresentationintheMaxineVM, last visited 20th April 2012.

[3] Sun Microsystems, "Whitepaper: Memory Management in the Java HotSpot™ Virtual Machine." http://www.google.co.uk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&ved=0CEUQFjAB&url=http\%3A\%2F\%2Fjava.sun.com\%2Fj2se\%2Freference\%2Fwhitepapers\%2Fmemorymanagement_whitepaper.pdf&ei=xrSfT7_HJoSi4gS-u-yRAw&usg=AFQjCNH1c7omTG339FJwX2-R_PBlgMgGzA&sig2=Zro257Wlb3qmb1aZBR3wtg, April 2006. last visited 1st May 2012.

[4] A. H. Richard Jones and E. Moss, *The Garbage Collection Handbook. The Art of Automatic Memory Management*. CRC Press, 1 ed., 2012.

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

[6] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Commun. ACM*, vol. 3(4), pp. 184–195, April 1960.

[7] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein, "An efficient parallel heap compaction algorithm," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pp. 224–236, ACM, 2004.

[8] R. R. Fenichel and J. C. Yochelson, "A LISP garbage-collector for virtual-memory computer systems," *Commun. ACM*, vol. 12, pp. 611–612, 1969.

[9] G. E. Collins, "A method for overlapping and erasure of lists," *Commun. ACM*, vol. 3, pp. 655–657, December 1960.

[10] S. Dieckmann and U. Holzle, "The allocation behaviour of the specjvm98 java benchmarks," in *Performance evaluation and Benchmarking with Realistic Applications* (R. Eigenman, ed.), pp. 77–108, MIT Press, 2001.

[11] D. M. Ungar, "Generation scavenging: A non-disruptive high-performance storage reclamation algorithm," in *SDE 1 Proceedings of the first ACM SIGSOFT/SIGPLAN software*

engineering symposium on Practical software development environments, pp. 157–167, 1984.

[12] "Source code monitor for the Maxine VM project." http://www.ohloh.net/p/maxine, last visited 29th April 2012.

[13] "Home - Maxine VM - Oracle Wiki." https://wikis.oracle.com/display/MaxineVM/Home, last visited 16th April 2012.

[14] "Boot Image - Maxine VM - Oracle Wiki." https://wikis.oracle.com/display/MaxineVM/Boot+Image, last visited 16th April 2012.

[15] "Maxine Bootstrapping - Self Organizing Systems Group." http://www.so.in.tum.de/wiki/index.php5/Maxine_Bootstrapping, last visited 16th April 2012.

[16] "Maxine Threads and Safepoints - Self Organizing Systems Group." http://www.so.in.tum.de/wiki/index.php5/Maxine_Threads_and_Safepoints#Transition_between_Thread_States_at_Safepoints, April 2012.

[17] "C1X-Maxine VM-Oracle Wiki." https://wikis.oracle.com/display/MaxineVM/C1X, last visited 16th April 2012.

[18] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 169–190, ACM Press, Oct. 2006.

[19] "DaCapo Benchmarks Home Page." http://dacapobench.org/, last visited 30th April 2012.

Appendix A

A.1 Building and Running the Maxine VM

The purpose of this document is to guide one user through the process of building and running the Maxine Virtual Machine.

A.2 Requirements

In order to run the virtual machine one needs to use the mx.py script located in $MAXINE_HOME/mxtool. The script requires an installed version of Python 2.7 and the $JAVA_HOME environment variable to be set to point to a jdk 6 / 7 installation (export JAVA_HOME=$JDK_PATH

A.3 Building and Running MaxineVM

The Maxine VM can be build either through the command line or from eclipse.

A.3.1 Command Line

Maxine can be built and ran from the command line as follows:

- Set up a helper script named for example “mx” in e.g. /usr/bin so the mx command can be used instead of mxtool/mx every time the VM is ran.

```
MX Helper Script.
#!/bin/bash

dir=`/bin/pwd`
while [ ! \( -f "$dir/mxtool/mx.py" -a -f "$dir/mx/projects" \) ]; do
    dir="$(dirname $dir)"
    if [ "$dir" = "/" ]; then
        echo "Cannot find 'mxtool/mx.py' in `/bin/pwd` or any of it's parents"
        exit 1
    fi
done

cd $dir;
exec mxtool/mx "$@"
```

- Run `mx clean` to clean up previous builds
- Run `mx build` to build the Maxine Source code (both Java and C files);
- Build the boot image of the VM. Run `mx image` (with the additional parameter: -heap=com.sun.max.vm.hea to use the Generational Heap Scheme)
- `mx helloworld` to test that the VM was set up correctly.
- Problem. On some operating systems, such as OpenSuse the environment variable LD_LIBRARY_PATH must be set up as well before running the VM to $MAXINE_HOME/com.oracle.max.vm.native/generated/li (location of the boot image).
- To run any program on the VM , simply use the command `mx vm foo`.

A.3.2 Eclipse

Maxine VM can also be developed from an IDE such as Eclipse. The Maxine VM project needs more heap memory than the default eclipse provides, therefore modify the file "eclipse.ini" by adding the following lines:

- -XX:MaxPermSize=512m
- -Xmx1g
- -Xms512m

To build the C files from eclipse as well the CDT plugin must be installed. Newer versions have taken out the eclipse project setting from the version control, so it might be neccesarry to run `mx eclipseinit` before importing the project in Eclipse. After importing the project, the build process should be done automatically. The boot image generation can also be ran from eclipse as follows:

```
Run => Run Configurations => Java-Application => BootImageGenerator-Default.
```

In the arguments field of the Boot Image Generator Default introduce the following string: "-heap=com.sun.max.vm.heap.sequential.generational" in order to run the virtual machine with the Generational Garbage Collector