FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Exact Garbage Collection for the Cacao Virtual Machine

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering & Internet Computing**

eingereicht von

**Michael Starzinger**
Matrikelnummer 0306126

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 22.06.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

# Exact Garbage Collection for the Cacao Virtual Machine

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Master of Science**

in

**Software Engineering & Internet Computing**

by

**Michael Starzinger**
Registration Number 0306126

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, 22.06.2011

(Signature of Author) (Signature of Advisor)

# Erklärung zur Verfassung der Arbeit

Michael Starzinger
Bacherplatz 7/3, A-1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit (einschließlich Tabellen, Karten und Abbildungen), die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22.06.2011 _____

(Unterschrift Verfasser)

# Abstract

Every virtual machine capable of executing Java byte-code needs to perform some sort of automatic memory management, commonly referred to as *garbage collection*. However, it is not specified how this garbage collection process actually has to be performed. So far the Cacao Virtual Machine (CacaoVM) used a *conservative* garbage collector that was designed for uncooperative environments, thus it has been unable to take full advantage of the information already present in the runtime infrastructure. This conservative garbage collector was replaced by an *exact* one, which has been tailored to the specific needs of a Java Runtime Environment.

In the context of Java several different components of the virtual machine need to be orchestrated for exact garbage collection to work. The application threads act as *mutators* on the heap and need to be prevented from interfering with the collector. Another central aspect is identifying the actual references which point into the heap and prevent objects from being collected. These references are referred to as the *root-set* and keep objects alive. The actual collection *algorithms* are in charge of cleaning up dead objects and reorganizing the heap, so that free memory is made available again. Those algorithms have already been researched in depth over the past decades, exhibiting vastly different characteristics.

All of the above aspects need to be considered from the perspective of a pure Just-In-Time compilation approach as taken by CacaoVM. Some crucial simplifications, which rely on fallbacks into an unoptimized interpreter mode, cannot be applied and need to be revisited in this setting. Special care was taken to allow future development towards a *generational* garbage collection approach by providing the necessary infrastructure.

# Kurzfassung

Jede abstrakte Maschine, die Java Bytecode ausführen kann, muss eine gewisse Art der automatischen Speicherverwaltung durchführen, welche als *Speicherbereinigung* bezeichnet wird. Jedoch ist nicht spezifiziert wie genau diese Speicherbereinigung umgesetzt werden muss. Bisher hat die Cacao Virtual Machine (CacaoVM) eine *konservative* Speicherbereinigung, welche für nichtkooperative Umgebungen entwickelt wurde, verwendet und konnte daher Informationen aus der Laufzeitumgebung nicht voll ausnutzen. Dieser konservative Ansatz wurde durch einen *exakten* abgelöst, der auf die speziellen Bedürfnisse einer Java Laufzeitumgebung zugeschnitten werden konnte.

Im Zusammenhang mit Java müssen diverse Komponenten der abstrakten Maschine aufeinander abgestimmt werden, damit exakte Speicherbereinigung funktionieren kann. Alle Ausführungsstränge einer Anwendung agieren als *Zugriffsmethoden* auf den Speicher und müssen davon abgehalten werden die Speicherbereinigung negativ zu beeinflussen. Ein weiterer zentraler Punkt ist das Auffinden von Referenzen, die in den Speicher zeigen und Objekte vor einer Bereinigung bewahren. Diese Referenzen werden als *Wurzelmenge* bezeichnet und halten Objekte am Leben. Die eigentlichen *Algorithmen* zur Bereinigung müssen tote Objekte aufsammeln und den Speicher so umstrukturieren, dass freie Speicherbereiche wieder zur Verfügung stehen. Diese Algorithmen wurden in den letzten Jahrzehnten tiefgreifend erforscht und weisen extrem unterschiedliche Eigenschaften auf.

Alle zuvor angesprochenen Aspekte müssen aus dem Blickwinkel einer puren Just-in-time-Übersetzung, wie sie von CacaoVM durchgeführt wird, betrachtet werden. Manche entscheidenden Vereinfachungen, welche auf ein Zurückschalten in einen nichtoptimierten interpretierenden Modus beruhen, müssen diesbezüglich erneut untersucht werden. Es wurde besonders darauf geachtet, eine Weiterentwicklung in Richtung einer *generationellen* Speicherbereinigung zu ermöglichen, indem die nötige Infrastruktur zur Verfügung gestellt wurde.

# Contents

CHAPTER 1

# Introduction

*Introduces the general concept of* garbage collection *in the context of Java and presents the scope of this thesis focusing on an exact and generational approach.*

> This junk isn't garbage! I can dig in any random pile and find something great.
>
> PHILIP J. FRY

## 1.1 Motivation

Modern computer languages no longer just only provide means to compile an application written in that language into machine code. They furthermore provide a complex runtime environment which the application links against and which helps in performing certain management tasks during runtime. This runtime environment can be considered an additional abstraction layer decoupling the application from the operating system, thus increasing portability. One of these tasks is *memory management*, that is, keeping track of which memory regions are in use by the application at the moment and which regions are free to be used when new memory is needed.

There are two fundamentally different approaches towards memory management. The first being *explicit memory management*, which leaves the task of allocating and freeing memory up to the application itself. The runtime environment merely provides functions for performing those two operations. The pseudo-code in listing 1.1 shows a typical pattern. An object is allocated using the `new` keyword. Once the object is no longer needed, it has to be explicitly freed using the `delete` keyword.

This example already illustrates the only two basic flavors of operations necessary to implement memory management. The simplest (but inefficient) implementation would be a direct mapping of these operations to their operating system pendants.

1. **Allocating memory**: Makes a continuous region of memory available to the application. Of course that region is exclusive to the requesting application and not allowed to overlap with other regions. See POSIX's `malloc()` for details.

```
Node createTree() {
    Node n = new Node();
    Leaf l = createLeaf();
    if (l == null) {
        delete n;
        return null;
    }
    n.addLeaf(l);
    return n;
}
```

Listing 1.1: Environment with explicit memory management

2. **Freeing memory**: Returns (or deallocates) a previously allocated region of memory to the runtime environment. It can then be reused when new memory is needed. See POSIX's `free()` for details.

The second, vastly different approach is *automatic memory management*, which delegates part of the aforementioned task of managing memory to the runtime environment. To be more precise, it delegates the task of freeing memory, whereas allocations are still done explicitly. The pseudo-code in listing 1.2 exemplifies this approach. Rather than having an explicit keyword to free objects, the runtime environment is in charge of determining when and how to free the object.

```
Node createTree() {
    Node n = new Node();
    n.addLeaf(createLeaf());
    return n;
}
```

Listing 1.2: Environment with automatic memory management

Comparing the two snippets of code above, the advantages of the automatic over the explicit approach should become evident [15].

- The risk of introducing memory-leaks is drastically reduced. It is no longer necessary for the developer to keep track of allocated objects on every possible path of execution.
- The risk of accidental freeing of memory still referenced by pointers to it, commonly called a *dangling pointer bug*, is completely eliminated.
- Source code readability is improved by focusing on the essential functionality instead of memory management issues.

- Uncommon cases where the machine actually runs out of memory are implicitly handled by *exceptions* instead of explicit `null`-pointer checks.

- Other types of resources might be coupled with the lifecycle of objects, which in turn would increase the overhead of management code even further. Those resources could be released together with the object itself. This technique is called *finalization* and will be discussed in detail in section 2.8.

Of course these advantages come with a certain price, because memory has to be freed at some point, it is merely done implicitly and in background by the *virtual machine* [15].

- Computational power is spent on deciding whether objects can be freed or not, whereas the application might have implicit knowledge of the answer.

- Memory usage might be inefficient and not take advantage of *memory locality* which leads to an effect called *thrashing* [13].

- The details of *garbage collection* might interfere with regular application execution and unexpected pause times or bad throughput are unwelcome side effects [26].

In the context of Java we deal with an object-oriented language executed inside a *virtual machine*. So the basic mean to allocate memory is by instantiating new objects [20, section 2.4.6]. The lifecycle and layout of these objects are described in section 1.2 and section 2.8 in detail.

## 1.2 Basics and Terminology

Realizing automatic memory management as motivated in the last section is the responsibility of the *virtual machine*, it has to free previously allocated objects which are no longer used by the application. The subsystem in charge of performing this task is called the *garbage collector* and it *reclaims* unused objects that reside in the garbage collected *heap*.

The decision whether objects are alive and need to be kept allocated, or dead and can be reclaimed, can basically be broken down to a graph-theoretical question of whether an object is *reachable* or *unreachable* for the executed application. This will be illustrated later on in section 2.1. This binary distinction of reachability can be further extended by introducing different strengths of reachability as discussed in section 2.8 and required by Java.

There are three tasks a *garbage collector* has to perform during a collection. Those tasks can be combined, performed separately or concurrently and even be split into several subtasks. But it is important that all three tasks are perfectly synchronized with other application or service threads being executed inside the same *virtual machine*. Those threads are commonly just referred to as *mutators*, because from the *garbage collector* point of view they mutate the heap while he tries to clean it up [15].

1. Determine *root-set* of pointers into the collected heap.

2. Distinguish between objects being *alive* or *dead*.

3. Reclaim memory of *dead* objects.

The rest of this section will discuss several general ideas towards accomplishing those tasks and the general ideas and terminology behind those approaches.

### Conservative Garbage Collection Approach

The *conservative* approach lacks the ability to correctly distinguish between reachable and unreachable objects. In case of ambiguity it conservatively sides on objects being reachable. This approach is easier to implement and has less implications for the overall system because not all references have to be clearly identified. The *garbage collector* can always resort to "guessing" to determine whether any value is a reference or not. However it has several major limitations.

- **Memory Waste**: Some non-reference values might look like references to the *garbage collector*, when in fact they are actually just of a primitive type. In this case more objects are kept alive than are actually reachable by the application.

- **Fragmentation**: Objects are pinned to their current location because the *garbage collector* is unable to update references, it might erroneously update a primitive value which was interpreted as reference. Hence free space is more and more split into smaller non-continuous parts. Precious free space is wasted due to *fragmentation*.

This is also the approach that CacaoVM has taken for the past years as further explained in section 1.3 below. It is the scope of this thesis to overcome the limitations of said approach.

### Exact Garbage Collection Approach

The *exact* approach can correctly distinguish between reachable and unreachable objects. There are mechanisms in place which allow to exactly locate all references which point to objects on the heap. Such an approach is far more complicated to integrate into a *virtual machine*, the whole chapter 2 deals with the implications of such an approach on CacaoVM or any other *Java Virtual Machine* for that matter.

Some further classifications might apply to implementations following this approach. The following is a short explanation together with an indication whether our implementation falls into that classification or not [15].

- **Moving Objects**: If a *garbage collector* has the ability to move objects around inside the heap, he is referred to as *moving*. Note that changing the location of an object implies updating all references pointing to that object. All algorithms presented in chapter 3 are moving ones.

- **Concurrent Collection**: The simplest way to synchronize *garbage collection* tasks with mutators is to suspend all mutator threads at safe points, perform all collection tasks and then resume all threads again. Such a procedure is called *stop-the-world* and section 2.5

explains our implementation of it. However, some *garbage collection* tasks might be executed concurrently with mutators, which would drastically reduce pause times. Especially on multi-core machines, suspending all threads is a very expensive operation performance-wise. However concurrent collection is outside the scope of this thesis [26].

- **Reference Tagging**: If the *virtual machine* would otherwise not be able to identify references or such a process would be too cost-intensive, references can be *tagged*. One bit of each value is used to mark references in-place. This has the advantage that references can be identified without further information lookup which preserves memory locality. On the other hand it has the drawback that primitive values are reduced to 31 respectively 63 bit length, so the *virtual machine* has to use *boxing* for values which need that bit for their representation.

### Generational Garbage Collection Approach

The *generational* approach is a special case of a moving and exact *garbage collector*. It separates the collected heap into regions, so called *generations*, which hold objets of roughly the same age. New objects are all allocated in the youngest generation (the so called *nursery* or *eden*) and over time moved (or *promoted*) to older generations until they reach the oldest (the so called *tenured*) generation [5, 15].

In figure 1.1 one can see a visualization of a heap consisting of three generations, left being the youngest and right being the oldest one. This approach makes sense under the following fundamental assumption which is believed to be true for almost all application scenarios [29].

> Most allocated objects will die young.
>
> *– Weak Generational Hypothesis (1)*

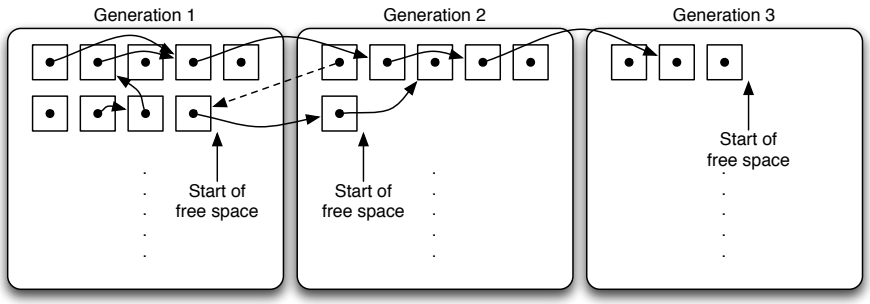



Figure 1.1: Generational approach towards garbage collection

The generational approach of composed heap regions has several advantages over one big continuous heap. The following is an overview of those advantages in the context of several aspects such a *garbage collector* bears with it [15].

- **Minor Collection**: The youngest region is the one filling up most often. Once it runs out of free space, it has to be evicted so that further allocations can succeed. Those minor collections happen more often but take much less time because only one generation has to be collected instead of the whole heap.

- **Efficient Copying**: Those minor collections require promotion of objects from one generation to the next. As we will see in section 3.1, those collections can be done very efficiently which further reduces performance-critical pause times for the mutators.

- **Fast Allocation**: The youngest generation (in which allocations occur) is never fragmented, there is definite address separating allocated from free space. Allocation can be implemented by simple *pointer bumping* of this address [5], hence it can be implemented very efficiently and even be inlined into the JIT-code.

- **Major Collection**: Only in the uncommon case that the oldest generation fills up, does the *garbage collector* run a major collection. For the last generation there is no target region to copy objects into, so other techniques (see section 3.6 for one example) have to be utilized.

There is one implication of the generational approach concerning backwards references (those from older to younger generations, dashed arrow in figure 1.1) which could be considered a drawback. This implication is discussed in section 4.3 as possible future work. Fortunately there is another fundamental assumption indicating that those backwards references are uncommon [29].

> Few references from older to younger objects exist.
>
> *– Weak Generational Hypothesis (2)*

As a conclusion of this section we believe a generational, moving and exact *garbage collector* to be the solution most worthy of aspiring towards for CacaoVM as a *Java Virtual Machine*. All following chapters aim towards that direction.

## 1.3 Current Situation in the Cacao Virtual Machine

All previous *garbage collector* implementations in CacaoVM were conservative ones at best. The implications that *exact garbage collection* has on the overall system were never acknowledged and the interfaces needed to be adapted accordingly. There were two usable implementations of collected heaps available at the time this thesis started.

### Non-collecting Heap

The first one is basically just a fallback for testing purposes, performing no collection at all. The heap consists of one continuous block in which *pointer bumping* is performed. Once this block is filled up, the *virtual machine* is permanently out of memory and can no longer allocate new objects.

### Boehm-Demers-Weiser Garbage Collector

The second implementation uses the so called BoehmGC library to manage the collected heap. It's a mature and highly optimized conservative *garbage collector* which can be integrated non-intrusively into most C (and C++) programs [8, 30].

To understand why BoehmGC can be used to manage the memory of a *virtual machine* despite of all the drawbacks mentioned in the previous section, we have to elaborate on some specific optimizations.

- **Efficient Free-List**: An efficient management of free-lists together with optimized locking provide fast allocation of memory [25]. Any *exact garbage collection* approach aiming to replace BoehmGC will have to eliminate locking completely to compete performance-wise. One possible approach to achieve that is outlined in section 5.1 as future work.
- **Grouping by Size**: Objects of same size are grouped into one region of the heap. This reduces fragmentation because reclaimed space between objects has roughly the same size as objects which will be allocated in that space later on.
- **No Intra-Object Pointer**: All references which would point into an object rather than to object boundaries are discarded. This drastically minimizes the number of false positives during pointer recognition.
- **Finalization**: There is support for *finalization* (see section 2.8 for details) in BoehmGC. This allows implementation of finalization support as requested by the Java specification [20, section 2.17.7].

The scope of this thesis is to investigate the necessary frameworks to support future *exact garbage collection* development in CacaoVM and lay ground for features which depend on such a *garbage collector*. The reference implementations presented in the process can in no way compete with the engineering work that went into BoehmGC.

CHAPTER 2

# Garbage Collection Infrastructure

*Describes the necessary infrastructure inside the* virtual machine *to support* exact garbage collection *and explains how those needs were satisfied in CacaoVM.*

Now, the hard part...

HIRO NAKAMURA

## 2.1 Discovering References into the Heap

Before a *garbage collector* can determine the reachability of objects on the heap, it has to find all references pointing into the heap from the outside. The set of such references is called the *root-set* as mentioned in section 1.2 before. Keep in mind that this *root-set* has to be complete. It has to be impossible for an application to reach an object inside the heap, which cannot also be indirectly reached through at least one reference in the *root-set* [15].

Note that references stored as part of an object will never point outside the heap. Any reference can either contain the `null` value or a valid reference to any object on the heap. There might be some cases where this constraint doesn't seem to apply.

- **Unsafe Memory Access**: There are some methods as part of the `sun.misc.Unsafe` class, which allow direct allocation of memory. However these addresses are passed as `long` values and hence are not recognized as references.
- **Direct Byte Buffer**: There is one type of object as part of the NIO interface, namely instances of `java.lang.nio.DirectByteBuffer` which handle memory outside the heap. Again, the actual address is stored as a `long` value.
- **Class Information**: Unfortunately CacaoVM stores instances of `java.lang.Class` outside the heap. The details behind that are discussed in section 5.4. But it should be the goal to move those objects into the heap and let the *garbage collector* manage them.

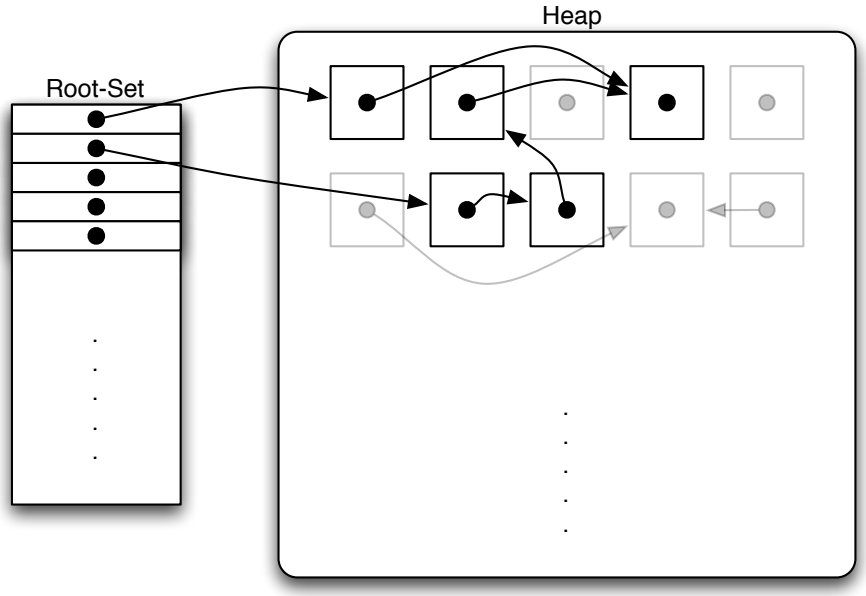



Figure 2.1: The *root-set* spanning a directed graph inside the heap

- **String Literals**: Same applies for `java.lang.String` instances which are loaded through string literals. Those should also be moved onto the heap in the future.

With these assumptions we can see that the *garbage collector* can traverse the directed graph spanned by the roots residing in the *root-set* as depicted in figure 2.1. It is hence possible to clearly distinguish between *reachable* and *unreachable* objects, the latter ones being grayed out in this figure. The algorithms through which this is accomplished are described in section 3.1 and section 3.3 later. For the sake of simplicity, most depictions of objects in this thesis just contain one reference marked by the black dot in the middle. But the concepts are trivially extendable to multiple references per object.

Now to the matter at hand, which subsystems of a *virtual machine* need to be considered in order for the *root-set* to be complete?

- **Machine Registers**: The actual processor registers might contain references into the heap used in the operation being executed by the current thread. It is not feasible for the *virtual machine* to remember the register allocation layout at each possible point in the code. But section 2.5 will show that a small subset of possible points in the code will suffice.

- **Thread Stacks**: Each thread of execution has a stack attached to it. This stack may contain stack-frames of JIT-code or native code. For JIT-code we know the exact layout and can locate all references (see section 2.6 for details). Unfortunately we have no way of knowing the exact layout of stack-frames for native code and need to rely on other mechanisms (see section 2.2 for details) [2].

- **Global References**: These are references which can be globally accessed by any thread, most commonly stored in static variables. The *garbage collector* provides service methods to register (or unregister) any location which contains such a global reference. Common places where such global references need to be registered are:
  - All static variables in the *virtual machine* codebase.
  - Global references registered through the Java Native Interface.
  - Objects held by the locking subsystem.
  - Objects used as *class loaders* registered in some class cache.
- **Local References**: For stack-local references in native code that are versatile enough to be passed to third-party native libraries, the so called *local reference* is used. The Java Native Interface defines functions for managing those references, but they are also used by the *virtual machine* internally (details are discussed in section 2.2 and section 2.4).
- **Thread-Local Information**: Each thread can store thread-local information inside an associated structure (similar to the `java.lang.Thread` object in the Java world). At the moment we just register each reference inside said structure as a global reference and be done with it.

Now that all the areas that could possibly hold references belonging to the *root-set* are identified, the *garbage collector* has to ensure that their respective exact locations can be determined at every possible point in the code, or that collections are only done while threads are suspended at such points. Basically the rest of this chapter deals with that central issue.

## 2.2 Direct vs. Indirect References

So far we have assumed that the *garbage collector* can exactly locate all references in machine registers and on a thread's stack. This might be true for JIT-code that the *virtual machine* generated, because it knows the exact types of each location. Unfortunately threads will also execute native code. Firstly CacaoVM itself is compiled using an ordinary C/C++ compiler which produces custom stack-frame layouts. Secondly Java supports loading of third-party native libraries at runtime which could have been compiled by any compiler. So it is virtually impossible to know the stack-frame layout of native methods.

The only feasible option of extending exact *garbage collection* to native methods as well, is to not pass direct references to those methods. So instead of direct references pointing into the collected heap, so called *indirection cells* or *handles* are used [1]. The illustration in figure 2.2 shows how those handles are used in CacaoVM.

All references passed to native methods are actually pointers into a table containing direct references into the collected heap. Thereby the entries in that table form indirection cells. By adding this table to the *root-set* the exact location of each direct reference (indirectly used by native methods) is known and can also be updated. Both, handles and local references follow this layout.

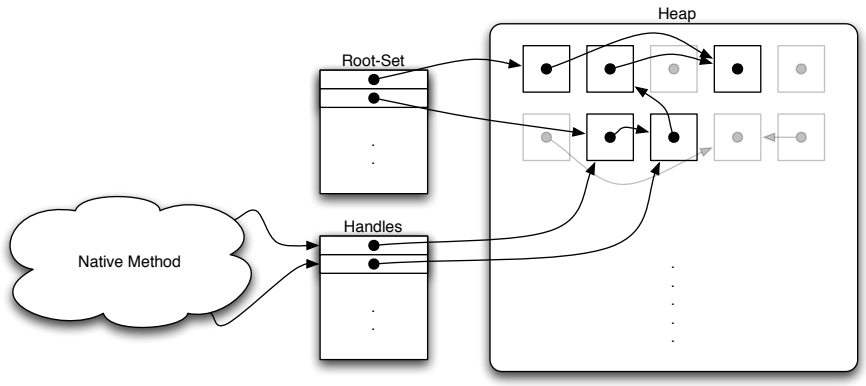



Figure 2.2: Usage of indirection cells for native methods

Of course we introduced one evident drawback, each object access in native code needs to take the indirection and furthermore synchronize with the *garbage collector* in some way. However we will show in section 4.1 that the associated runtime penalty can be considered negligible for real-world applications.

### Access through the Java Native Interface

Fortunately the Java Native Interface (JNI) was already designed with handles in mind [19]. All references passed into third-party native methods are stored in so called *local references*. Previously in CacaoVM those were equivalent to direct references, but now they are equivalent to handles. All actual accesses to objects on the heap are done through functions provided by the JNI as listing 2.1 exemplifies.

```
JNIEXPORT jint JNICALL
Java_Test_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
    jsize len = (*env)->GetArrayLength(env, arr);
    jint *data = (*env)->GetIntArrayElements(env, arr, 0);
    for (int i = 0; i < len; i++)
        sum += data[i];
    (*env)->ReleaseIntArrayElements(env, arr, data, 0);
    return sum;
}
```

Listing 2.1: Sample of array access via JNI inside a native method

This sample native method returns the sum of all integers in an array. The JNI function for getting the array elements actually returns a copy of the array data, thereby ensuring the underlying object can be moved by the *garbage collector* if needed. One major overhaul of the JNI implementation in CacaoVM allowed us to realize all such changes.

Another solution would be to *pin* the object inside the heap until it is released by the appropriate JNI function again, thereby making it temporarily unmovable. But since our *garbage collector* implementation does not support temporal *pinning* of any kind, we don't use this approach.

### Access through the internal interface

Using the above interface for internal operations inside the *virtual machine* as well would be too inefficient, so there is a second internal interface to access objects on the heap. This interface is sometimes referred to as the *low-level native interface* and has to synchronize with the *garbage collection* as well [1].

Whenever an indirection is taken and a concurrent movement of the referenced object is not allowed, the thread performing that access is said to be inside a *critical section*. The typical access of an object on the heap performs the following steps.

1. **Enter Critical Section**: The fact that a thread is inside a critical section is represented as a boolean flag inside it's thread-local information. Flipping this flag is efficient, because each thread only modifies it's own flag, so no synchronization is necessary. The *garbage collector* only reads those flags when threads are suspended, so no race-condition emerges.

2. **Take Indirection**: Taking the indirection cell returns a direct pointer to the referenced object's current position. As long as the critical section is active, the *garbage collector* won't move the object.

3. **Perform Access**: Direct access to the object's content can be performed using the direct pointer from the previous step. Primitive values can be used as-is, whereas reference values have to be wrapped into indirection cells themselves before the critical section ends.

4. **Leave Critical Section**: The diametrically opposite operation to the first step, same reasoning as above applies in this case.

5. **Poll for Pending Collection**: A global flag indicates whether the *garbage collector* requested a stop-the-world while the thread was inside the critical section, in which case the thread will suspend itself. In section 2.5 the details and reasoning for this step are discussed.

To perform the above steps efficiently without cluttering the source code of CacaoVM, we introduced so called *accessor classes*. These are C++ classes performing typed access on existing handles for given Java classes through the use of inline methods. In listing 2.2 one example of copying the content of a string is shown.

This example takes a handle for an instance of `java.lang.String` and copies it's content into the given buffer. Three of the object's fields (namely `value`, `count` and `offset`) are accessed through the accessor class. Another array accessor class performs the actual copy operation of a region from the string value. All the details discussed before are hidden inside those accessor classes.

```
void string_copy(Object* string, char* buffer, int size)
{
    java_lang_String jls(string);
    Object* value  = jls.get_value();
    int32_t count  = jls.get_count();
    int32_t offset = jls.get_offset();

    CharArray ca(value);
    ca.get_region(offset, MIN(count, size), buffer);
}
```

Listing 2.2: Sample of array access via internal interface inside a native method

## 2.3 Efficient Storage of Indirection Cells

Whenever a reference value is read in native code, it is wrapped inside a newly created handle. Therefore creating and managing handles has to be implemented as efficiently as possible. Another aspect not discussed so far is the ability to destroy handles when the referenced object is no longer needed in native code.

To solve both issues we implemented a so called *handle memory*, which is a thread-local growable data structure which can be freed in bulks. To understand what that means, let us look at those characteristics in detail.

- **Thread Locality**: All handles are by principle thread-local, they are only used by the thread who created them. For an efficient implementation this is extremely important because it permits to avoid locks altogether. Whenever any thread wants to share a reference with another thread in native code, those references have to be made into *global references* as outlined in section 2.1 before.

- **Growable Structure**: Since there is no upper boundary on the number of references a thread can hold at a moment, the data structure has to be capable of holding an indefinite number of entries. This is achieved by separating it into blocks, the so called *handle memory blocks*, which in turn form a single linked list with it's head being the most recent block.

- **Fast Creation**: As creating handles is the only common operation, it has to be as fast as possible. The location of the next available entry inside the most recent block is known, we just use the entry at that location and bump the pointer forward. There is no fragmentation inside the blocks.

- **Freeing in Bulks**: Having to free each handle individually would nullify the advantages of a *garbage collector* in native code. So handles can only be freed in bulks. This is realized by placing markers, the so called *handle memory borders* inside the data structure. At a certain point (e.g. at method entry) the marker is placed at the current position. Later on

(e.g. at method exit) the most recent border can be popped, thus destroying all handles created after the marker was set.

With the functionality of this *handle memory* described above, it is possible for native methods to create their own scopes inside of which handles can be used efficiently. Once those scopes are left, handles are automatically destroyed and the referenced objects can be collected by the *garbage collector*. Consider figure 2.3 for a depiction of how this data structure looks like.

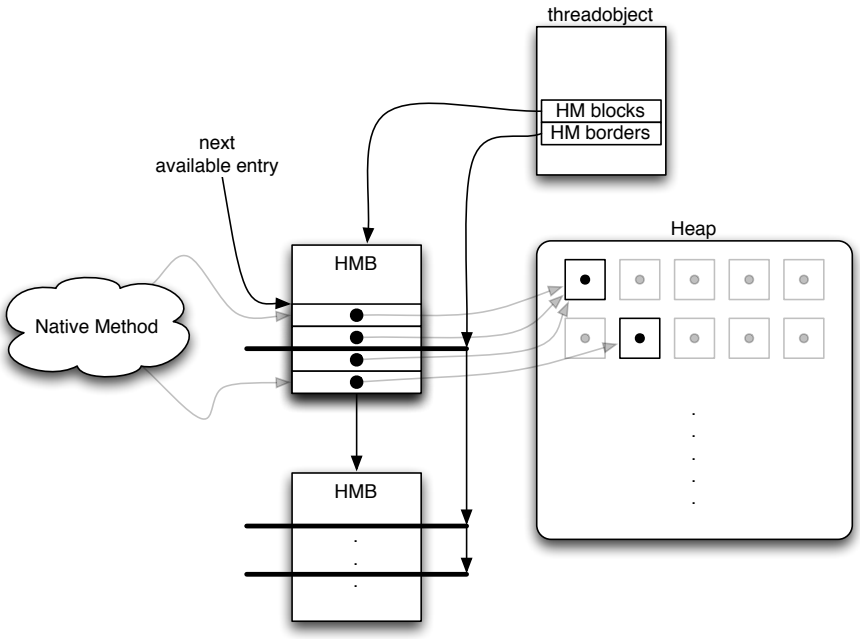



Figure 2.3: Design of the handle memory data structure

The only issue left, now that we have an efficient way of managing handles, is to identify those places in the *virtual machine* where *handle memory borders* should be placed. As a rule of thumb, each support function that is called regularly and internally makes heavy use of handles should have it's own border (e.g. all JNI functions). Enough borders have to be placed, so that long-running methods do not keep objects alive unnecessarily.

## 2.4 Modes of Execution inside the Virtual Machine

The previous sections talked about JIT-code as well as native code and how differently those two types have to be treated. Each thread executed inside the *virtual machine* always executes code which is of one of those two types. Commonly JIT-code originates from Java methods being executed, whereas native code comprises of supporting functions of the *virtual machine* that were compiled using a C/C++ compiler.

As section 2.5 will point out, it is very important to know the exact mode of execution each thread is currently in, to allow threads to be suspended correctly. Therefore special care has to

be taken for transitions between modes to be well-defined to eliminate race-conditions. A flag as part of the thread-local information is used to indicate the current mode of execution, the following sections will describe the transitions changing this flag.

In order to be able to implement those transitions in a way that is almost platform-independent we reused the concept of an *execution-state* which was introduced by the *on-stack replacement* implementation. The *execution-state* represents the machine state (i.e. processor registers) and can be read out of the machine or written back into it. The listing 2.3 contains the actual structure used to represent said state [28].

```
struct executionstate_t {
    uint8_t   *pc;                          /* program counter */
    uint8_t   *sp;            /* stack pointer within method */
    uint8_t   *pv;                         /* procedure vector */
    uint8_t   *ra;         /* return address / link register */

    uintptr_t  intregs[INT_REG_CNT];       /* register values */
    double     fltregs[FLT_REG_CNT];       /* register values */

    codeinfo  *code;     /* codeinfo corresponding to the pv */
};
```

Listing 2.3: Platform-independent representation of the machine state

### Native-to-Java Transition

This transition is used to enter JIT-code generated by the *virtual machine* itself. As outlined in section 2.2, direct references are used inside JIT-code, hence all references wrapped inside handles passed into it need to be unwrapped. The following ordering of transition operations ensures the transition is free of race-conditions.

1. **Enter Java World**: Set the thread's mode of execution to `java` before any direct references into the heap are created. This prevents concurrent movement of referenced objects.

2. **Prepare Machine State**: Unwrap handles and prepare an *execution-state* containing direct references. Should the thread be suspended here, the mode is already set to `java` but the JIT-code cannot be determined because the machine didn't yet jump into it. So the race is detected.

3. **Write Machine State**: The transition is finished by finally writing the prepared *execution-state* into the machine. Thereby the program-counter will point into the JIT-code, effectively jumping into it.

Upon return the above steps are reversed bottom-up so that the thread can end up in the calling native code again.

### Java-to-Native Transition

This transition is diametrically opposite to the previous one, hence all references passed into the native code need to be wrapped into indirection cells. But on top of that, the Java Native Interface specification requires us to use *local references* for those indirection cells [19]. The following steps ensure a safe and correct transition.

1. **Read Machine State**: The transition starts by reading the machine state and preparing an *execution-state* to contain all values, including direct references.

2. **Prepare Local References**: Wrap all references into *local references* (as mentioned in section 2.2 they have the same layout as handles). Thereby those references can either be used by the Java Native Interface or by the internal interface, depending on how the underlying native code was designed.

3. **Exit Java World**: Now that all references are safely wrapped, the thread's mode of execution can be set to `native` and the thread can safely execute native code.

4. **Push Handle Border**: As a courtesy to the callee and as a precaution, we install the first handle border right away. This destroys all handles created inside the native code even if the callee forgets to use handle borders and prevents possible handle leaks.

Again upon return the above steps are reversed bottom-up so that the thread can end up in the calling JIT-code again.

### Fast Intrinsics Transition

There are some supporting native methods frequently called from within JIT-code, which are referred to as intrinsics (or *builtins* in CacaoVM-speak). Doing the full Java-to-Native transition roundtrip for them would be inefficient, hence we introduced the concept of *fast-builtins* to CacaoVM as an optimization.

Those fast intrinsics use direct references although they are actually native code. Should a thread be suspended while inside a fast intrinsic, the *garbage collector* will detect this because no associated JIT-code can be found. To avoid deadlocks all fast intrinsics must have the following characteristics.

- Don't block and return "fast".
- Don't throw exceptions out of the intrinsic.
- Don't allocate any objects which would cause collections.

Even for intrinsics not having those characteristics the optimization could be applied by splitting the intrinsic into two parts. One full implementation for the *slow-path* and one partial implementation just providing the *fast-path* and falling back to the full implementation if necessary. Remember that the fast-path is a perfect candidate for inlining into JIT-code.

The framework for supporting this two-fold approach was introduced into CacaoVM, and was so far only utilized to experiment with *lock-inlining*. But it most certainly provides more optimization potential.

### Trap Handling Transition

For dealing with uncommon cases of faults (e.g. exceptions, compiler-invocations, replacement points, ...) the concept of *traps* is used in CacaoVM. In a POSIX environment those traps are delivered by the operating system in the form of *signals* that the *virtual machine* handles. Together with those signals a machine-dependent state information (i.e. the *machine-context*) is passed as well.

Since traps can only occur in JIT-code and all the handling functions are realized as native code, a transition similar to the *Java-to-Native* case discussed above needs to be performed. The only major difference to the necessary steps outlined there, is that we use conversion functions to convert a *machine-context* into an *execution-state* and vice-versa in this case.

As a general word of caution, keep in mind that the trap-handling code is very complex and fragile. Even though it might seem straightforward at first glance, think over every change to it twice or thrice, because *you will break stuff*.

## 2.5 Thread Suspension Mechanisms

As mentioned before, our *garbage collector* implementation uses a *stop-the-world* approach. This requires threads to be suspendible regardless of their current state of execution in a reasonable amount of time. Since CacaoVM knows two different modes of execution (as illustrated in section 2.4) which have to be handled separately, this section will explain how to handle both.

Even semi-concurrent approaches of *garbage collection* require some sort of thread suspension, so having a working framework for that purpose is necessary for future development and testing as well. But keep in mind that we only introduced a reference implementation which is not yet fully optimized.

The notion of *suspending a thread* in this context not only means preventing the thread from being executed concurrently, but also making sure it is suspended at a well-defined point inside the code at which it's state can be examined (and possibly changed) [1]. Those points are mostly referred to as *safe-points* in literature. Our implementation uses the following steps for initially suspending a thread and later performing a *roll-forward* depending on it's mode of execution.

1. **Flag Pending Collection**: Set the global `gc_pending` flag indicating that a *garbage collection* is pending and hence a suspension at *safe-points* is requested.

2. **Suspension by Signal**: Send a dedicated signal to every initialized thread, thereby interrupting it's current execution. The appropriate signal handler will store the current *execution-state* so that it can be used to unroll the thread's stack in section 2.6 later. It is important to wait for all signal handlers to finish their job to ensure the stored *execution-state* is accurate.

3. **Determine Mode of Execution**: According to the current mode of execution of every thread, a different suspension approach might be taken. Reading the appropriate flag unsynchronized is safe, because the thread mutating it is suspended at the moment.

After the suspension mechanism for each thread finished and all threads have successfully acknowledged their suspension at a *safe-point*, the *garbage collector* can finally commence with unrolling each thread's stack as section 2.6 will describe.

### Suspension in Native Mode

At this point the thread exclusively uses indirection cells as introduced in section 2.2 to represent references into the heap. Hence suspending it is safe in most cases, expect for *critical sections* which are rather short and uncommon [1].

4. **Check for Critical Section**: In case any of the indirection cells are taken, the thread will indicate so by setting the appropriate thread-local *critical section* flag. Reading this flag unsynchronized is safe, because the thread mutating it is suspended at the moment.

5. **Roll-Forward**: Should the thread be inside a critical section and hence be unsafe to suspend, it will be resumed. The global `gc_pending` flag makes sure the thread will suspend itself as soon as the critical section is left. This explains why critical sections need to be reasonably small in order for the suspension to be efficient.

### Suspension in Java Mode

At this point the thread might hold direct references in machine registers and on it's stack, which need to be located exactly. Instead of assuming every position in the code is safe expect those inside critical sections, the opposite approach is taken. It's generally assumed the current position in the code is unsafe and some *roll-forward* to the next well-defined *safe-point* is required.

4. **Determine the Code**: Internal information for the code that the thread is currently executing needs to be found. This is achieved by either looking up the *procedure vector* or the *program counter* in the current *execution-state*. In case no such information can be found, the thread is in transition between modes of execution as explained in section 2.4. The current solution in this case is to resume the thread for some time and retry suspending it again, which clearly is a rather sub-optimal solution in need of optimization.

5. **Activate Traps**: By activating (or *arming*) traps at *safe-points* inside the current code, the thread will again suspend itself after being resumed. We reused the trapping mechanism already present from our *on-stack replacement* implementation. There need to be enough traps to ensure the thread will not escape the current code and will suspend in a finite amount of time, so traps are placed at the following instructions [28].

    - All method invocations of any kind.
    - All backwards branches (which would form loops).

- All return statements (which would escape the code).

6. **Roll-Forward**: Resume the thread and rest assured, that it will suspend itself in a reasonable amount of time through a hook in the *on-stack replacement* subsystem.

## 2.6 Unrolling a Thread's Stack Information

One central aspect of *exact garbage collection* is to exactly locate all references a thread's machine state and stack holds. This has to be done for all threads running inside the *virtual machine* and is generally referred to as the process of *walking the stack*. Note that it is closely related to how and where threads have been suspended (see section 2.5 for details).

In general each stack in CacaoVM can hold two types of stack-frames, so called *native frames* and *JIT frames*, arbitrary interleaved and held together by some glueing frames. Basically for each transition described in section 2.4 some sort of glueing frame is required. The layout of these frames is depicted in figure 2.4.

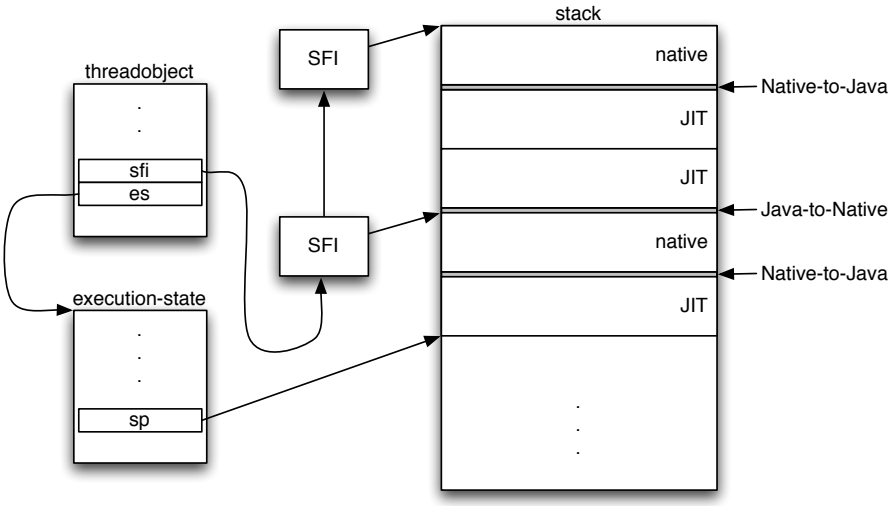



Figure 2.4: Layout of a thread's stack

The process of *walking the stack* is very similar to stack-unwinding as done when exceptions are thrown. All stack-frames are successively popped from the stack (to be more precise, the popping is simulated) until the last frame is reached. Additionally, locations of references into the heap are gathered to fill the *root-set*. This is all done in close cooperation with the *on-stack replacement* implementation because of it's similar functionality [28].

- **Popping JIT Frames**: The exact layout of the topmost JIT frame is known, because the thread was suspended at a *safe-point*. All successive JIT frames are also in the state of a *safe-point* because they are at the point of an invocation of some sort.

- **Reverse Native-to-Java**: The glueing frame of a Native-to-Java transition can be identified because of it's unique *program counter* value, which causes the popping of JIT frames to stop.

- **Popping Native Frames**: The content of a native frame is completely unknown to the *virtual machine* because it's layout follows that of the compiler which produced the underlying native code. No assumptions about that layout can be made. However all references are wrapped in indirection cells as section 2.2 pointed out. The previously recorded *stack-frame information* (SFI) allows us to pop the native frame (it might actually contain more than one frame, but the *virtual machine* doesn't care about that) and continue at the next glueing frame. The whole process stops in case the last *stack-frame information* object is reached.

- **Reverse Java-to-Native**: The glueing frame of Java-to-Native invocation allows reentry into the popping loop. It also contains callee-saved registers which are needed to restore an accurate machine-state at that point.

At each step during this process the *execution-state* for the given thread is transformed to represent the machine state at each stack-frame level. All references contained within registers are represented by an entry in that *execution-state* and can also be added to the *root-set*.

This was a very simplified description of the general approach taken in CacaoVM and only outlines the general cornerstones of the concept. A lot of corner cases appear in conjunction with the *trap handling transition* outlined in section 2.4. *The devil is in the details.*

### Layout of JIT Frames

As stated several times, the layout and type information of machine state and stack is known for all points in the code. At first it is just known at *compile-time* but needs to be preserved for *runtime* at well-defined points.

The infrastructure to preserve the type information was already provided by the *on-stack replacement* mechanisms through so-called *replacement points* [28]. However that type-information actually is too precise and hence introduces an unnecessary overhead when used solely for *garbage collection* purposes. The *garbage collector* does not need precise type information for registers and stack-slots, it just needs a binary distinction between reference and non-reference values.

Therefore we propose an optimized *garbage collection point* which contains a subset of the information present in a *replacement point* and can be used when such a point is used solely by the *garbage collector*. The proposed implementation contains two bit-fields of fixed upper lengths to represent said binary distinction for registers respectively stack-slots. Should the size of the stack-frame exceed the upper bound, the fallback is to simply use a regular *replacement point* instead. This optimization will reduce the memory-consumption by necessary management information drastically.

## 2.7 Object Identity Hash-Codes

One important concept of object-oriented programming is that every initialized object has it's own identity. Consequently it is always possible to check whether two given references refer to the *same* object. Note that this is inherently different from checking whether two (possibly different) objects are considered *equal* in the context of an application. The distinction between those two types of comparisons should become evident when considering the `String.equals()` method.

In an environment without moving *garbage collection* the identity comparison is equivalent to a pointer comparison. Actually the generated JIT-code performs a simple pointer comparison in the moving *garbage collection* setting as well as the listing 2.4 illustrates. However it is important to keep in mind that these pointers are highly volatile because objects are allowed to move.

```
======== L000 ========
IN:  <null> javalocals: [La0(rdi) La1(rsi)]
 134:   0:  ALOAD           L0 => La0(rdi)
 134:   1:  ALOAD           L1 => La1(rsi)
 134:   2:  IF_ACMPNE       La0(rdi) La1(rsi) --> L002
 134:   3:  NOP
OUT: []

 134:
0x00007ffff7e8fde0:   cmp    %rdi,%rsi
0x00007ffff7e8fde3:   jne    0x00007ffff7e8fdf8
```

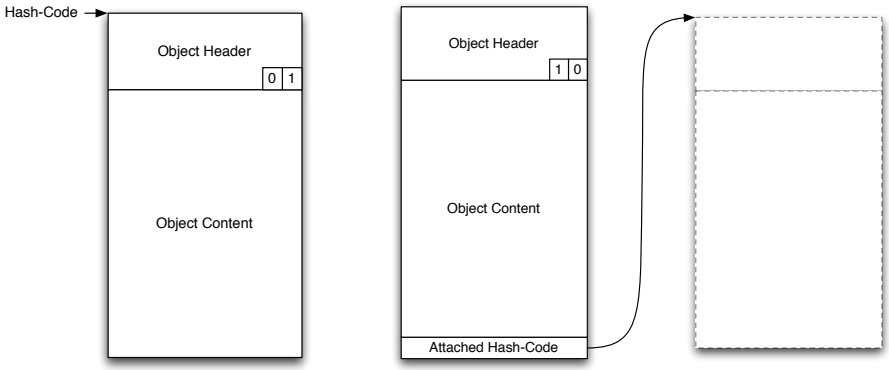
Listing 2.4: Object identity comparison in JIT-code

At first glance the realization of the object identity concept inside a *virtual machine* seems straightforward, however problems arise when using the object identity to hash objects. This happens for example when objects are put into hash-tables without overriding the appropriate `Object.equals()` and `Object.hashCode()` methods.

The typical implementation in a non-moving environment is again to use the actual pointer value as the hash-code. Note that most hashing-based algorithms are also optimized towards hash-codes having pointer characteristics (i.e. aligned at 4-byte boundaries). But it is of central importance that the hash-code of an object remains unchanged over time, even after the *garbage collector* has moved the object on the heap.

One solution is for the *virtual machine* to also return the pointer value as hash-code but for the *garbage collector* to attach the initial hash-code to the object once it is moved around on the heap [6]. This value can be attached to the end of an object, causing its actual size to increase by one word (32 or 64 bit). This is possible for every move operation discussed in chapter 3 because the target location is always big enough.

The *virtual machine* considers three different states any object can be in, when it comes to hashing object identities or moving objects on the heap. A combination of two identity-bits




(a) Object after hash was taken for the first time but before it has been moved.

(b) Object with original hash-code attached to it's end after it has been moved.

Figure 2.5: Attaching hash-code to object

inside the object header is sufficient to keep track of those states. The illustration in figure 2.5 shows how these identity-bits correspond to attached hash-codes.

- **Normal**: The object identity was not yet taken, the object has not been hashed. In this state the *garbage collector* can freely move the object on the heap. (Bit-mask: `00b`)

- **Taken**: Once the application queries the hash-code and hence knows the current value, that hash-code has to be preserved so that all consecutive queries return the same value. As long as the object does not move, it can remain in this state. (Bit-mask: `01b`)

- **Attached**: Once the *garbage collector* decides to move an object for which the hash-code was taken and hence is known to the application, it has to record the original value and attach it to the object. (Bit-mask: `10b`)

Note that one optimization technique for *virtual machines* is to provide intrinsics for commonly used methods like `Object.hashCode()` which can then be inlined. Otherwise those methods would be backed up by native implementations which are very hard (if at all possible) to inline. Our slightly more complicated implementation of said method does increase the complexity of such intrinsics, but they could still be inlined if needed.

For the sake of completeness, let's also note that using a separate table to keep track of attached hash-codes is no feasible solution, because efficient implementations of such a table would require correct object identity hashing themselves, leading into a typical chicken-and-egg problem.

## 2.8 Different Reachability Strengths in Java

In addition to the two basic states (*reachable* and *unreachable* as described in section 2.1) that an object can be in from a *garbage collection* point of view, Java knows several other interim states between those two. This section will discuss those states, their meaning and possible use-cases.

The runtime system provides so called *reference objects* which are treated specially by the *garbage collector* and themselves refer to one object, the so called *referent*. The following is a list of possible strengths together with some details about the appropriate reference object from strongest to weakest. The actual reachability is defined by the strongest link leading to the *referent* [23].

- **Strong Reference**: Any usual reference inside a Java application that does not utilize a *reference object* is a strong one. This is equivalent to the notion of *reachable* as used so far.
- **Soft Reference**: Objects that are *softly reachable* are eligible to be reclaimed at the discretion of the *garbage collector*. The heuristic used to reach that decision is not specified, but softly reachable objects need to be reclaimed before an `OutOfMemoryError` is thrown. They are commonly used to implement application-level caches.
- **Weak Reference**: Objects that are *weakly reachable* will be reclaimed. The weak reference acts just like a soft one, only that there is no heuristic involved in the decision whether an object should be kept alive or not. They are commonly used to associate required cleanup tasks with objects without keeping them alive.
- **Finalization**: In case an object implements the `finalize()` method, it will be *finalized* exactly once before being reclaimed. Note that the finalizer might resurrect an object and therefore prevent it's reclamation, but that doesn't change that invariant.
- **Phantom Reference**: Objects that are *phantomly reachable* have been finalized but not yet reclaimed. In contrast to soft and weak references, all phantom references will be dealt with after the referent has been finalized. The reference will not be cleared by the *garbage collector*, it is the responsibility of the application to make it eligible to be reclaimed. They are commonly used to associate required post-finalization cleanup tasks with objects.
- **Unreachable**: This is finally the reachability strength that condemns objects to be reclaimed once and for all, there is no way to prevent the *garbage collector* from doing so. This is equivalent to the notion of *unreachable* as used so far.

Almost all *reference objects* will be cleared once their respective reachability strengths apply and they might even be enqueued into a `ReferenceQueue`, if they are registered with one. This gives applications some level of control over *garbage collection* and the object life-cycle [23].

CHAPTER 3

# Methodology and Algorithms

*Presents the main* garbage collection *algorithms used by the collector and how they were orchestrated in two reference implementations as part of* CacaoVM.

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> DONALD E. KNUTH

## 3.1 Algorithm for Copying Collection

For the *generational garbage collection* approach to be beneficial, it is important to have an efficient algorithm for *promoting* live objects to the next generation, thereby evicting the current generation. This section presents one such so called *copying algorithm* which is based on Cheney's algorithm [9] and is used in one form or another in many *generational garbage collection* implementations [5, 26].

The general idea of this algorithm is to perform both tasks of finding reachable objects and moving them into another region in one single phase. The region being evicted is referred to as *source region* and the one where objects are copied into is called *destination region*. This is done by *forwarding* objects from the source into the destination region and leaving a forward marker in the source region pointing to the new location. The following two requirements on the underlying object layout emerge.

1. **Forward Pointer**: Each object has to be large enough to hold the forward marker pointing to it's new location. This can easily be satisfied because in CacaoVM each object contains the `vftbl` entry which can be misused for this purpose.

2. **Markable Pointer**: The forward pointer needs to be marked in some way to identify it and distinguish it from a regular `vftbl` value. Since all our *virtual function tables* are at least 4-byte aligned, the least-significant bit of that pointer can be used for marking purposes.

The central function of the algorithm is one performing the forwarding operation on objects. It maintains the `next` pointer, separating allocated from free space. As mentioned in section 1.2 this is part of the *fast allocation* principle in our regions anyways. The simplified pseudo-code in listing 3.1 describes the forwarding operation in detail [5].

```
forward(PTR):
  if (PTR points into source-region) then
    if (PTR[vftbl] is marked) then
      return unmark(PTR[vftbl]);
    else
      copy(NEXT, PTR, PTR[size]);
      PTR[vftbl] = mark(NEXT);
      NEXT += PTR[size];
      return unmark(PTR[vftbl]);
    endif
  else
    return PTR;
  endif
```

Listing 3.1: Pseudo-code of the forwarding operation for copying collection

Using this forwarding operation, a non-recursive copying collector can be implemented by maintaining a `scan` pointer which is initialized to the value of the `next` pointer. Forwarding references in the *root-set* and copying objects into the destination region (thereby making them alive) will advance the `next` pointer. However those objects haven't yet been scanned for further references themselves. By scanning objects in the destination region for references which in turn need to be forwarded, the `scan` pointer will be advanced until both pointers refer to the same memory address. At that point all references have been forwarded and all reachable objects have been copied into the destination region [15, section 7.1]. The figure 3.1 should further illustrate this procedure step-by-step for a simplified example.

This space- and time-efficient algorithm is perfectly suited for promoting objects between generations, assuming that the *root-set* for a particular generation contains all the references pointing into that generation. A certain caveat regarding this precondition is discussed in section 4.3. Also the destination region needs to have enough free space to hold all the life objects from the source region. But besides that the algorithm requires no additional data structures outside the heap and runs in linear time depending on the number of live objects in the source region, a fact that is particular helpful considering that most objects will die according to the *weak generational hypothesis* [29] as outlined in section 1.2.

## 3.2 Reference Implementation *Semi-Space Heap*

The first reference implementation is the so called *semi-space heap* and is solely based on the copying algorithm presented in the previous section 3.1. It divides the collected heap into two

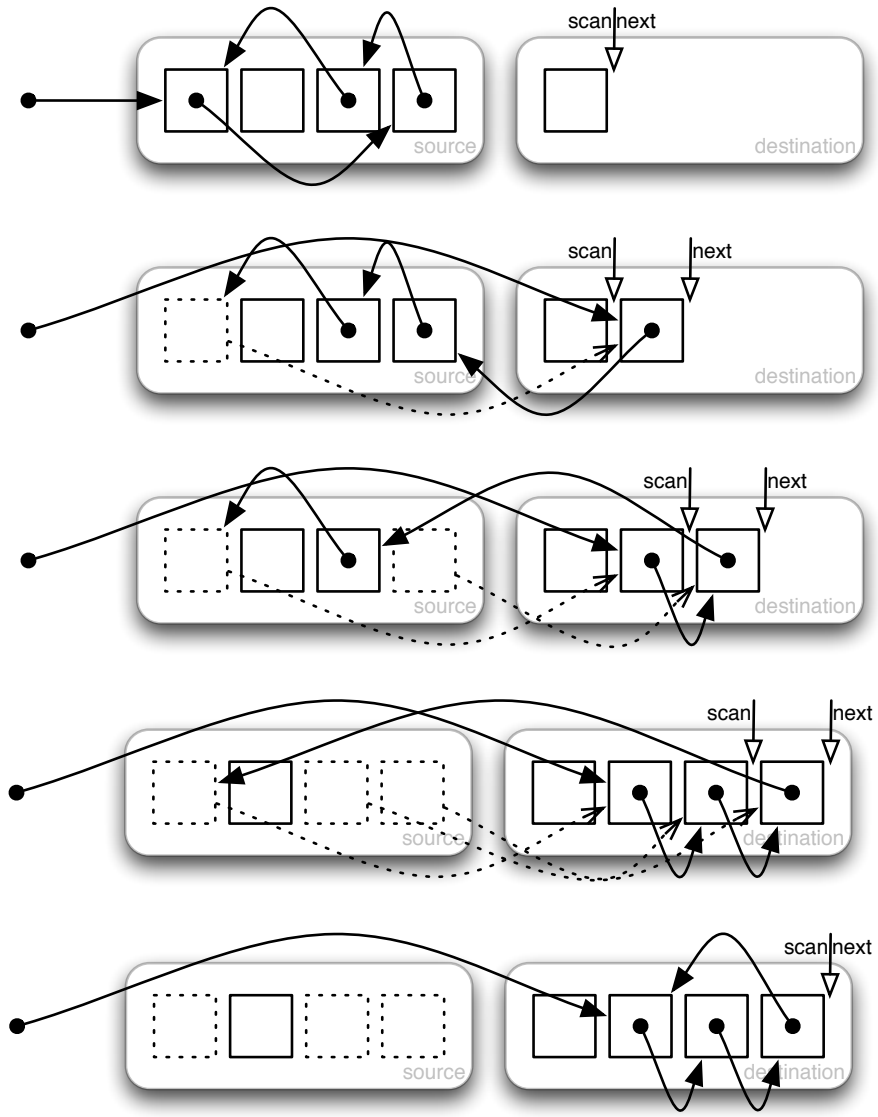



Figure 3.1: Illustration of the copying collection algorithm

equally sized regions, with only one being active at a time. The first one is called *from-space* and is the target for all allocations. The second one is called *to-space* and is kept empty. During the *garbage collection* all live objects are copied from the *from-space* into the *to-space*, after which those spaces switch roles [15].

The obvious hitch with this heap implementation is that only half of the heap is usable by the application at any time. Hence this heap implementation is barely usable in production-ready scenarios, but serves as a good test implementation for the copying algorithm. To further improve test coverage the *from-space* can be overwritten with *canary words* after each collection, so that all old object locations are destroyed and erroneous access can be detected.

## 3.3 Algorithm for Marking Live Objects

Before the *garbage collector* can reclaim unused memory, all live objects reachable from the *root-set* need to be marked to prevent them from being reclaimed. This process is referred to as *marking* or *tracing* and is commonly implemented as some sort of depth-first traversal through the heap. However a simple recursive implementation cannot be used, because deep structures on the heap would overflow the collector's stack, so a more efficient work-list has to be used.

The presented marking algorithm is based on Shorr's algorithm [27] which places the mentioned work-list on the heap itself to reduce the amount of memory used for additional data structures. There are basically two pieces of information required for each entry on the work-list, the first is a reference to the object being traversed, the second is the number of references in that object that have already been visited. The actual *marking bits* which indicate whether an object was visited or not are stored as part of the object header.

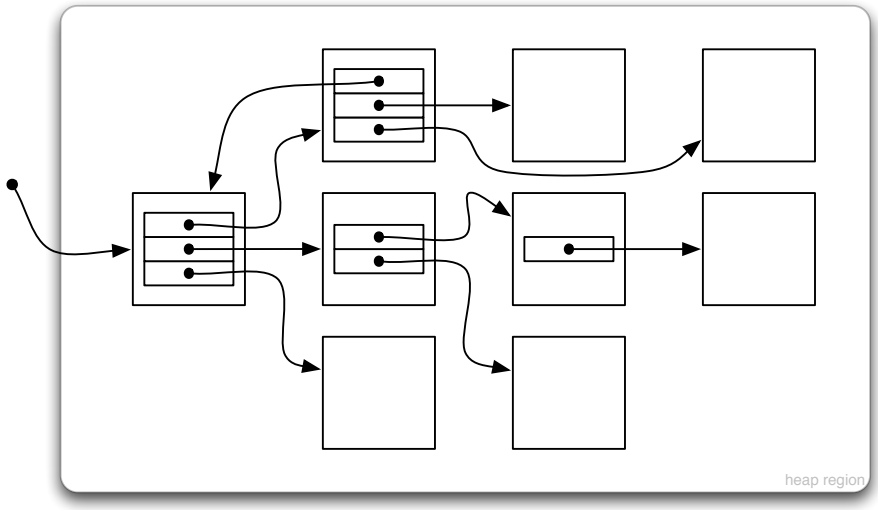
- **Object Reference**: The figure 3.2 illustrates how references to objects on the work-list are stored as a reverse linked list inside the objects themselves. The back-link to the previous entry (depicted as a dashed arrow) is stored in the reference currently being visited. If all references inside the current object have been visited, the back-link can be taken to backtrack to the previous level and the original value of said reference can be restored. References already visited are depicted gray whereas those yet to be visited are depicted white.

- **Reference Number**: The back-links actually point to the object being on the work-list. To find the reference inside that object to which the backtrack occurs, the number of references already visited needs to be stored. For small values this number can be stored inside the object header, for larger values we use an external stack as a fallback solution. For ordinary objects we store the field index (for arrays the array index) of the reference being visited at the moment.

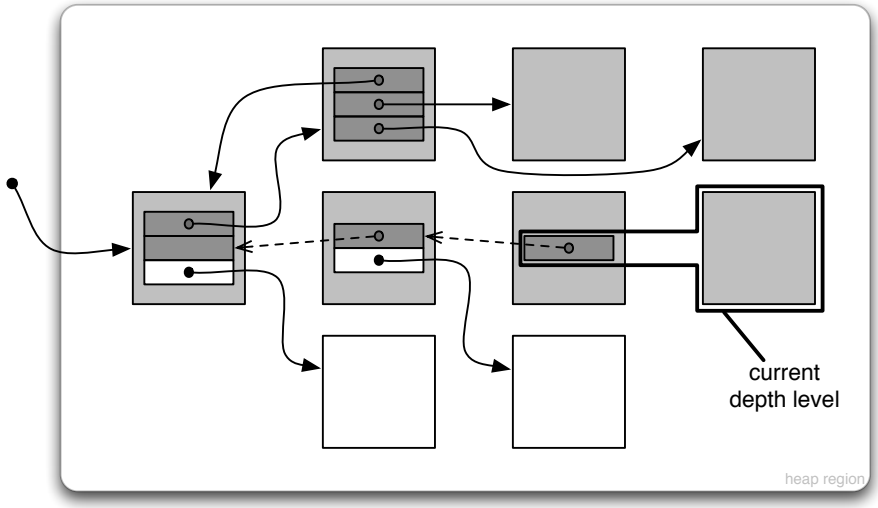
- **Marking Bits**: To actually mark objects as visited we use two *marking bits* inside the object header. If an object which has already been marked gets visited a second time, traversal of it's references will be skipped because it is either already on the work-list or has already been traversed completely. Objects which have been marked are depicted gray whereas unmarked objects are depicted white.

This space-efficient tracing algorithm only requires minimal data structures outside the heap to mark all reachable objects. However objects containing references and placed on the work-list are visited several times, associated class information needs to be looked up at each visit.

Also this algorithm only works with suspended mutators and is not suited for any sort of concurrent *garbage collection*, because the heap is left in an unstable state and modifications by mutators cannot be detected. For the implementation of concurrent approaches a tri-color marking algorithm needs to be used [11].




(a) Graph in original state before tracing algorithm has started.




(b) Graph in modified state containing the work-list while tracing.

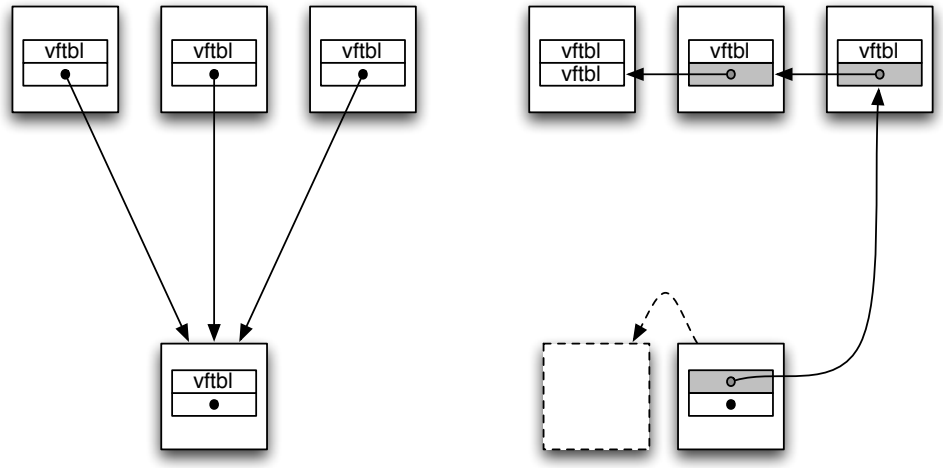
Figure 3.2: Illustration of tracing algorithm to mark live objects

## 3.4 Algorithm for Reference Threading

One important aspect when moving objects around the heap, is to update all references pointing to them accordingly. If the original memory location is not overwritten, a simple forward marker can be left (as presented in section 3.1) and references can be updated after the object has been moved. Unfortunately this solution is unfeasible if the original memory location might get destroyed.

Furthermore managing additional data structures outside the heap is not possible, because

*garbage collection* algorithms are executed at times when free memory is a sparse commodity. In this section an efficient algorithm for updating references is presented which is referred to as *reference threading* or *pointer reversal* [15, section 5.3].




(a) Graph in original state before references were *threaded*.

(b) Graph in modified state after references were *threaded*.

Figure 3.3: Illustration of threading algorithm to update references

In subfigure 3.3a the original graph before references are threaded is shown. Each of the three objects holds a references pointing to the target object to be moved. Similar requirements to the ones presented in section 3.1 apply, pointers need to be markable and one freely modifiable pointer needs to be available in each object. We use the same technique as outlined before to satisfy these requirements. Next the objects are processed from left to right. Each pointer to the target object is reversed so that `vftbl` is marked and points to the location of the actual reference (not the object holding it). The original value of `vftbl` is preserved in the reference itself.

In subfigure 3.3b the final graph after successively reversing all three references is shown. All references are threaded into a single linked list of marked references (depicted in gray) with the head in `vftbl` of the object to be moved and the tail being the first un-marked pointer. Now it is possible to move the target object to a new location and update all references to it by simply following that linked list and un-threading each reference. The tail of said list will contain the original `vftbl` value of the target object which can then be restored.

This space-efficient algorithm for updating references to moved objects requires no additional data structures outside the heap. The threading of a references is done in constant time and un-threading of all references is done in linear time depending on the number of threaded references. It is used in CacaoVM by the compaction algorithm presented in section 3.5.

## 3.5 Algorithm for Compacting Memory Regions

Most *generational garbage collection* approaches require some sort of non-copying algorithm for major collections, to reclaim memory in the oldest generation as well. The algorithm presented here is of the *compacting* type and moves all live objects to one end of the region, thereby eliminating all fragmentation caused by interleaved dead objects.

The presented compaction algorithm is based on Morris' algorithm [22] which originally required three phases. A major precondition is that all live objects must have been marked before, which is achieved by executing it after the marking algorithm presented in section 3.3 has been applied. The following illustration is for the original three-phase variant because it's easier to understand. The actual implementation uses an optimized two-phase variant which will be discussed below.

The general idea is to *thread* and update all references (as discussed in section 3.4 before) in the first two phases and only actually move objects in the last phase. This causes any destructive movement, which would break lists of *threaded* objects, to be done after all objects have been *unthread*. The following is a detailed description of those phases.

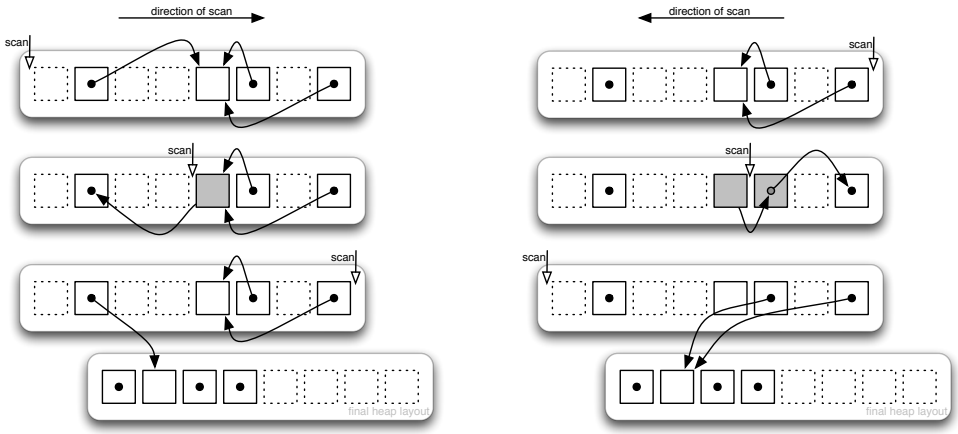
- **Preparation**: Thread all references being part of the *root-set*, thereby making sure those references are updated when objects are moved. Note that this step does not determine whether objects are alive or dead, because that was already done by a previous marking algorithm.

- **Phase 1**: Scan through objects in forward direction while threading and updating all forward references. Once a live object (identifiable because it's marked) is reached, all forward references to it have been threaded by now and can be updated by unthreading them. The final target location of the object is known by keeping track of the size of all live objects encountered so far.

- **Phase 2**: Scan through objects in reverse direction while threading and updating all backwards references. Same reasoning as for the previous phase, only in reverse direction, applies here. By the end of this phase all references point to the correct location. But since objects haven't been moved yet, they all point to a memory location of random content, so the heap is in a highly unstable state.

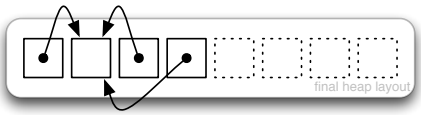
- **Phase 3**: Scan through objects in forward direction one last time and move each live object to it's respective target location. Note that this actually is a *move* operation as opposed to a *copy* operation, because source and target memory may overlap. However, objects are only moved into one direction, either they stay at the same location or move backwards because a dead object has been reclaimed. All references have already been updated by the previous phases and are now pointing to actual objects.

To further illustrate the necessary steps, figure 3.4 shows a simplified example. Objects being unmarked and hence dead, are depicted with dashed lines. References being part of a threading are depicted as gray objects. The final heap layout is drawn separately although it actually represents the same region.




(a) Region during phase 1 of compaction scanning forward and updating forward references.

(b) Region during phase 2 of compaction scanning reverse and updating backward references.




(c) Region after phase 3 of compaction moved objects to their respective target locations.

Figure 3.4: Illustration of the compacting collection algorithm

As mentioned before, the actual implementation uses an optimized two-phase variant of the algorithm [16]. This optimization is possible because the moving of objects can be merged together with the unthreading of backward references into one phase. Also scanning a heap containing objects of variable size in reverse direction, as the original variant requires, is actually a problematic requirement. The following is a description of the reduced set of necessary phases.

- **Preparation**: Same as in the original variant.
- **Phase 1**: Scan through objects in forward direction while threading and updating all references. However the unthreading operation in this phase will only update forward references correctly, all backward references will remain threaded, because they are encountered after the target object has been scanned.
- **Phase 2**: Scan through objects in forward direction while updating all backward references that are still threaded. Also move objects to their respective target locations. This can be done, because all references not updated so far are located in objects yet to be scanned.

This space- and time-efficient algorithm can be used for collections in the last generation of a generational *garbage collector*. However the notion of being time-efficient needs to be

relativized. The algorithm runs in linear time depending on the number of objects inside a region. To be precise it needs to scan trough all (living or dead) objects inside a region two times without even accounting for the pre-required marking.

## 3.6 Reference Implementation *Mark-and-Compact Heap*

The second reference implementation is the so called *mark-and-compact heap* and is a combination of the previously presented marking algorithm (see section 3.3) and the compaction algorithm (see section 3.5), as the name suggests. The collected heap consist of one big region which is compacted during each *garbage collection*, hence keeping it free of fragmentation and allowing fast allocation [5, 15].

The memory consumption of this heap implementation is far better as compared to the *semi-space heap*. Unfortunately the collection process has a bad runtime performance because of the complexity of the algorithms involved. One possible solution is a combination of both presented heap implementations as suggested in section 4.3. Again this heap serves as a test implementation for the involved algorithms and test coverage can be improved by overwriting free space inside the heap with *canary words* after each collection.

CHAPTER 4

# Results and Future Work

*Gives an overview of achieved results and pointers to possible future work towards the realization of a* generational garbage collection *approach.*

> Time is that quality of nature which
> keeps events from happening all at once.
> Lately it doesn't seem to be working.
>
> ANONYMOUS

## 4.1 Results and Comparison of Runtime Impact

Again it has to be emphasized that our *garbage collector* implementation does not intend to compete with BoehmGC performance-wise, but instead intends to provide infrastructure for *exact garbage collection* development. Hence we are not comparing the reference implementations presented in chapter 3, instead we compare the infrastructure changes presented in chapter 2 to the previous situation in CacaoVM. All measurements were taken on a quad-core `x86_64` machine running CacaoVM compiled against GNU Classpath, both in the most recent version at the time of writing.

In figure 4.1 the negative impact of several changes to the infrastructure of CacaoVM is illustrated using the CaffeineMark 3.0 benchmark scores (higher is better). Red bars represent the base implementation without any added modifications. One major modification causing a slight performance drop was the addition of execution mode transitions as introduced in section 2.4. Measurements after adding this change are depicted in green. Another major modification depending on the previous one was the introduction of indirection cells for native code as presented in section 2.2 and section 2.3. Measurements after adding both changes are depicted in blue.

Considering pure computational performance of generated JIT-code the impact can be considered negligible as the results for *sieve*, *loop* and *logic* show. The only major drop can be observed in the handling of operations related to string modifications and could be combated by further pursuing inlining of intrinsics as mentioned at the end of section 2.4.

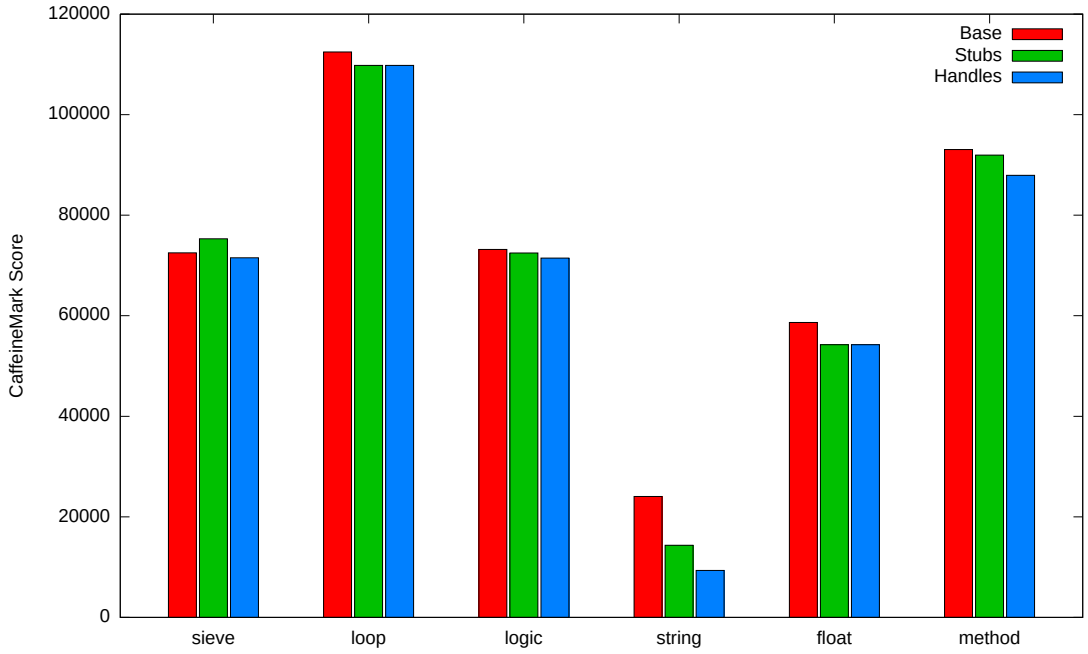



Figure 4.1: Measurements of performance impact by infrastructure changes

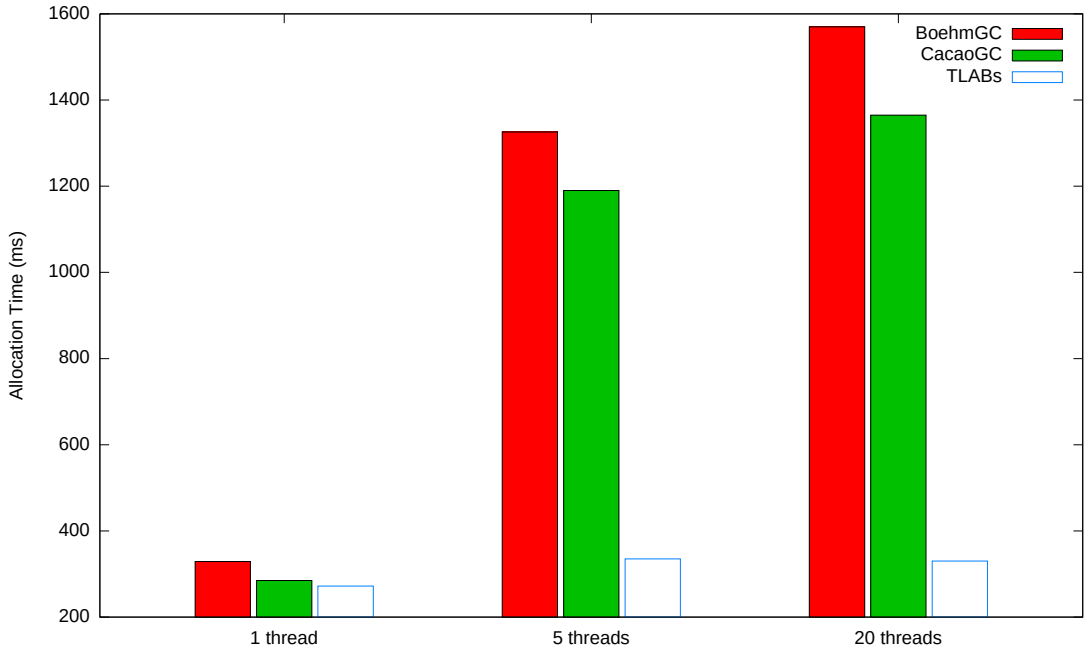



Figure 4.2: Measurements of allocation time for two million allocations

In figure 4.2 accumulated time for the allocation of two million objects is shown. Note that these numbers only contain runtime needed for allocation (lower is better), initialization as well as garbage collection time is not accounted for. The original implementation delegating to BoehmGC is depicted in red. Fast pointer-bumping in one locked global region (as used in section 3.2 and section 3.6) is depicted in green. An experimental prototype of thread-local allocation as presented in section 5.1 is indicated in blue. However this prototype is highly unstable because collections inside thread-local areas will almost certainly reclaim live objects. Each thread is given a fixed allocation buffer to perform unsynchronized allocations in. Once a buffer fills up it is completely evicted into the global heap, ignoring any old-to-young references. This prototype is able to bootstrap the *virtual machine* and execute the above allocation benchmark, any more complex application will crash due to live objects being falsely reclaimed.

Objects are concurrently allocated by the number of threads specified until a total allocation count of two million is reached. For the single-threaded mode all objects are allocated consecutively by a single thread, hence contention of the lock synchronizing the global heap never happens. For the multi-threaded modes that single lock experiences more and more load because all threads try to allocate concurrently, leading to permanent contention.

This shows that fast pointer-bumping is a bit more efficient than the allocation that BoehmGC can provide. Providing an inlined version of the allocation intrinsic would further improve allocation time. However both variants suffer from the effects of synchronization due to one global heap, as the thread-local prototype should clearly demonstrate. Especially multi-core machines will highly benefit from development towards that direction.

Note that this metric only shows pure allocation time, any additional overhead needed to evict thread-local areas is not taken into account by the above measurements. However there are two reasons why memory management tasks should be postponed as long as possible in this case. Most of the objects that are allocated thread-locally will die young and not survive the first promotion. Coupled with the computational complexity of the involved copying algorithms which solely depend on the number of *living* objects, such minor collections have low pause times.

## 4.2 Results and Comparison of Heap Usage

Another interesting measurement parameter is the size of used heap memory over time. For this purpose collections were triggered at regular intervals and the used heap space was calculated by subtracting reported free space from total heap size. Note that this measurement is independent from the actual memory consumption a *virtual machine* exhibits, because total heap size is not stated and the *garbage collector* might decide to increase or decrease total heap size at it's own discretion.

In general the *exact garbage collection* approach has a lower heap usage due to it's precise nature. The heap usage of the conservative approach can serve as an upper bound and is expected to be higher or equal the heap usage of an exact collector. This situation is exemplified in figure 4.3 using the DaCapo Eclipse Benchmark with a maximum heap size of 128 megabytes (option `-Xmx128M`) and forced collections each second, comparing BoehmGC against one of our implementations presented in section 3.2 and section 3.6. For this measurement those two

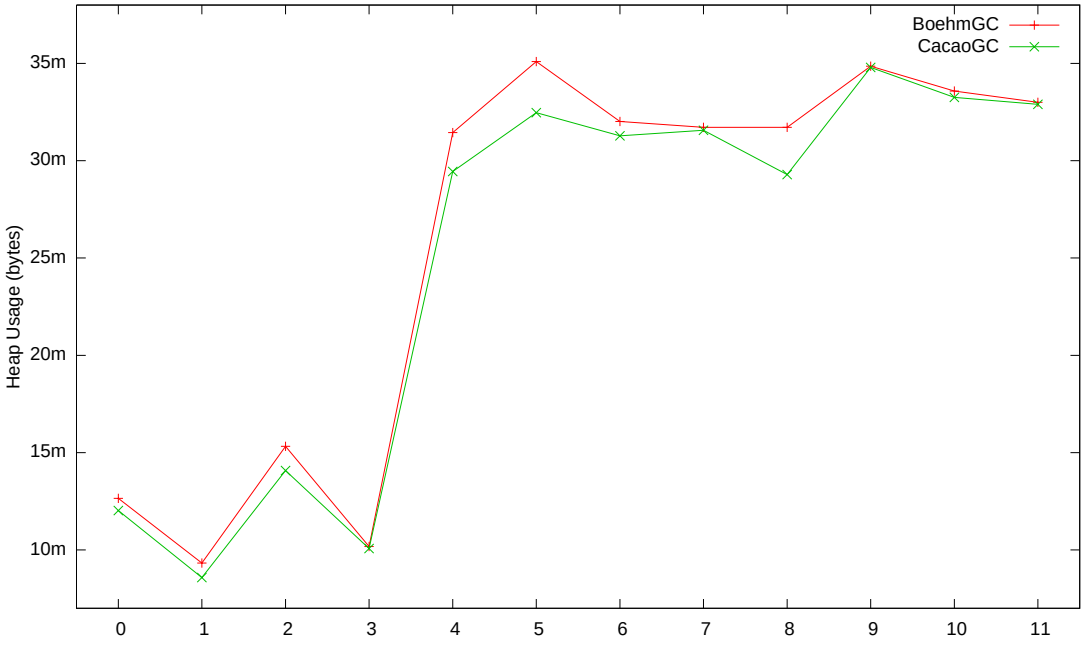



Figure 4.3: Measurements of heap usage over time

implementations can be used interchangeably, because they exhibit exactly the same degree of precision and even use the same codebase to calculate a *root-set*.

## 4.3 Missing Pieces for Generational Garbage Collection

One remaining issue to successfully deploy *generational garbage collection* in CacaoVM, which was just briefly mentioned so far, concerns references from older to younger objects. Even though these references are uncommon according to the *weak generational hypothesis*, they still need to be handled correctly. For an illustration of such *old-to-young references* see figure 1.1 in one of the earlier chapters. These references need to be added to the *root-set* for copying collection as described in section 3.1.

Aside from dedicated hardware-support there are several software solutions to keep track of those references, almost all based on so called *write barriers*. Such barriers are executed whenever a references inside the heap is updated (i.e. either through a field-store or an array-store operation) and checks if the new reference value would create a back-reference. The following is a list of common approaches towards implementing write barriers, which mainly vary in the granularity of the information they record [14].

- **Remembered Sets**: Each generation has an associated *remembered set*, which records all locations that might contain references into that generation from an older one. The finest granularity would be to record the reference location itself, but a more feasible solution

might just record the object containing the reference. During collections those locations recorded in the *remembered set* are scanned for references that have to be added to the *root-set* for that generation.

- **Card Marking**: By dividing each generation into smaller regions, so called *cards*, an even coarser granularity can be achieved. Each *card* can be marked as *dirty* in case it might contain old-to-young references. These marking bits are associated with the source-region of references as opposed to *remebered sets* which belong to the destination-region. During collections all *dirty cards* of all older generations are scanned for references pointing into the region being collected. Issues arise when objects transcend card boundaries, because finding object headers might not be possible in that case.

- **Page Protection**: One variant of the above *card marking* technique is to use page protection mechanisms provided by the underlying operating system to detect when pages are *dirtied*. In this case *cards* are equivalent with memory pages.

Coincidently these write barriers also help to support the implementation of concurrent marking techniques as briefly mentioned in section 3.3 and section 5.3, that require keeping track of concurrent reference updates by mutators [11, 26].

CHAPTER 5

# Conclusion and Related Work

*Explains design decisions previously presented for this exact* garbage collector *in comparison to other state-of-the-art* garbage collection *implementations.*

> I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.
>
> ABRAHAM H. MASLOW

## 5.1 Thread-Local Allocation

As mentioned in section 1.2 already, one goal of any memory management scheme should be to achieve *fast allocation*. All presented heaps as part of this thesis use simple *pointer bumping* for allocation, one of the most efficient methods of allocating memory, tuning it further seems virtually impossible. However there is one detail that hasn't been discussed so far, namely the required locking of the heap during allocation.

The heap is a global storage area that is accessed concurrently by all *mutators*, hence these accesses need to be synchronized. However some of the allocated objects will be used exclusively by one thread and don't have to be allocated on a global heap. This immediately suggests the introduction of thread-local storage areas for objects and would avoid any unnecessary synchronization during allocation, thus further increasing allocation speed.

### Locality through Escape Analysis

One approach to achieve above goal is to apply *escape analysis* onto the underlying code [21]. Thereby it can be proven that objects allocated at certain allocation sites will never *escape* a predefined scope.

- **Method Scope**: Objects that don't escape the scope of a method can be put directly onto the stack. No explicit allocation is necessary, however the underlying memory still has to be cleared.
- **Thread Scope**: Objects that don't escape the scope of a thread can be allocated on a *thread-local heap* without the need for any synchronization. Even collections can be performed completely without synchronization.
- **Global Scope**: No additional constraints can be bestowed upon objects allocated at sites with global scope, those still have to be allocated on a global heap.

The major advantage of using escape analysis to solve this issue is the distinction between several different scopes for allocation sites. Unfortunately escape analysis determines scopes for allocation sites and not for the actual objects themselves, thereby exhibiting conservative behavior for indecisive results.

### Locality through Generations

Another approach is based on the *generational garbage collection* infrastructure presented in this thesis and requires no additional static analysis of any kind. Assuming that most objects won't escape the thread scope, each object is by default allocated in a thread-local area of the heap acting as an own generation. This approach is very similar to using *thread-local heaps* and has the major advantage that again allocation can be done without any synchronization. Even collection can be done on a thread-local basis without suspending other mutators [12].

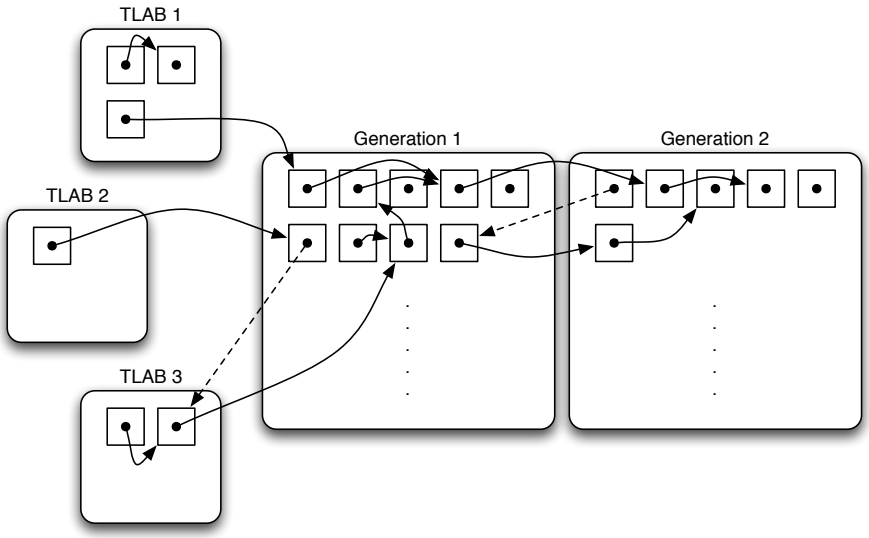



Figure 5.1: Thread-local allocation through generations

In figure 5.1 an illustration of these generations placed in front of the global heap, which are sometimes referred to as *thread-local allocation buffers* (TLABs), is shown. During the first

minor collection (which can mostly be done thread-locally as well) these buffers are evicted and promoted objects are placed onto the global heap. Thereby the generational infrastructure implicitly performs dynamic escape analysis as discussed in section 4.3. Again the *weak generational hypothesis* is the basis for this optimization. It can be shown that such an approach can even be faster than stack allocation [4].

## 5.2 Liveness Analysis of Local Variables

So far only two kinds of type-precision were considered when it comes to local variables that are stored as part of a thread's stack or machine registers. The conservative approach has no type information wheres the exact approach has precise type information for each local variable. However there are a total of four degrees of precision worth considering when it comes to the calculation of the *root-set* in a *Java Virtual Machine* [3].

1. **Ambiguous**: Every local variable is treated as a possible reference without considering it's actual type. This leads to *ambiguous root-sets* as they are used in conservative *garbage collection* covered in section 1.2. This degree of precision was previously implemented by CacaoVM by utilizing BoehmGC.

2. **Type-Precise**: Accurate type-information is used to obtain a *type-precise root-set* only containing reference types. Appropriate techniques to store and use such information are covered throughout this thesis and are the basis for *exact garbage collection*. This degree of precision is used by the presented *garbage collector*.

3. **Live-Precise**: Augmented information from an intra-procedural live variable analysis is used, to only add local variables which are alive and result in a *live-precise root-set*. Objects to which the method still holds a references, but which definitely will not be accessed again, can be reclaimed. This degree of precision is commonly used by modern *garbage collection* implementations [3].

4. **Refined Live-Precise**: More complex analyses, such as an inter-procedural live variable analysis, can be used to further refine precision of the *root-set*.

Note that determining exactly which local variables an application will access again is equivalent to the halting problem. Thus a fully-precise calculation of the *root-set* is not computable, but liveness analysis as presented here is a good approximation of that optimum.

Using a more precise *root-set* helps to further reduce memory usage. Some of this behavior can be simulated by explicitly setting references to `null` in an application. But this practice introduces unnecessary overhead, burdens the developer with memory management issues and on top of all, is unable to cover all cases of local variables losing their liveness.

## 5.3 Concurrent Garbage Collection

Large pause times caused by major collections in the oldest generation can be reduced by using concurrent techniques. The marking algorithm presented in section 3.3 requires all mutators to

be suspended. To realize concurrent marking of reachable objects, a tri-color marking algorithm can to be used [11].

Initially all objects are colored *white*, thereby indicating that they are unmarked. The marking algorithm then continues with tracing references and coloring referenced objects *gray*. Those objects have been marked to be reachable, but their references have not yet been scanned, hence they are still on the work-list. Once all references in an object have been scanned and it is removed from the work-list, it is finally colored *black*. This algorithm requires the so called *tri-color invariant* to hold throughout the marking phase. To ensure that said invariant holds, *write-barriers* as described in section 4.3 are used to return *black* objects that are concurrently modified, back to a *gray* state, in order for them to get rescanned again.

> During marking there will be no edge pointing from a black node to a white one.
> *– Strong Tri-Color Invariant*

Even though the original tri-color algorithm can achieve almost full concurrency with mutator threads, most implementations settle with so called *semi-concurrent* or *semi-parallel* algorithms for the sake of better throughput [7]. A typical implementation might use several short pauses for exclusive marking, interleaved with concurrent marking and sweeping phases [26].

- **Initial Marking Pause**: Calculate the *root-set* by *walking the stack* of each mutator thread while it is fully suspended as described in section 2.6.
- **Concurrent Marking Phase**: Perform the tri-color marking on said initial *root-set*, while mutators are being executed concurrently. After marking finishes, most objects are *black*, while some might still be *gray* because of concurrent updates to one of their reference fields. However the *write-barriers* described in section 4.3 record such updates.
- **Final Marking Pause**: Again calculate the *root-set* like in the first step, but also add recorded *gray* objects from the previous concurrent phase. This final marking step is executed with all mutators being suspended and ensures that all concurrent changes from the previous phase are correctly marked as well.
- **Concurrent Sweeping Phase**: A non-moving sweeping phase can be performed concurrently again, because it only deals with dead *white* objects which cannot be reached by mutators anymore. Compacting objects as presented in section 3.5 however cannot be done concurrently, because it modifies live objects as well.

This shows that *semi-concurrent* marking could be added to the presented *garbage collector* once *write-barriers* are in place. However the compaction of memory regions, which takes up a large portion of the pause time of major collections, cannot easily be parallelized and is outside the scope of this thesis.

Especially for large-scale applications with a high degree of parallelism deployed on multiprocessor servers, the aspect of minimizing pause times without loosing throughput becomes very important. Some *garbage collection* algorithms even allow to specify a *soft real-time goal* for pause times. This is often coupled with *incremental garbage collection*, which is very similar to the concurrent approach. The *Garbage-First* collector for example partitions the heap into regions, much like generations but unsorted. It uses *remembered sets* (see section 4.3) to record all locations that might contain pointers to objects within the region. Instead of sweeping dead objects or compacting live ones, the regions are always evicted (or *evacuated*) by copying live objects into another region as discussed in section 3.1 before. An arbitrary set of such regions can be chosen for collection, preferring the ones containing lots of dead objects, hence the name of that *garbage collector* [10].

## 5.4 Unloading of Class Information

At the moment CacaoVM stores all class information outside the heap. Every instance of `java.lang.Class` is un-collectable and lives outside the collected heap, which directly contradicts one of the fundamental assumptions in section 2.1. This circumstance requires special-casing in all presented collection algorithms and makes class unloading hard to implement.

The concept of *class unloading* is an optimization which helps the *virtual machine* to further reduce memory consumption. It has to be completely transparent to the application whether the underlying *virtual machine* supports it or not, therefore accidental reloading of classes (which would reset static variables and rerun static initializers) has to be prevented. This is realized by coupling the life-cycle of class information to the loading class loader. *A class or interface may be unloaded if and only if its class loader is unreachable. The bootstrap class loader is always reachable; as a result, system classes may never be unloaded. [20, section 2.17.8].*

One proposed solution to implement class unloading and collection of class information memory, by reusing most of the infrastructure for *generational garbage collection* presented so far, would have the following cornerstones.

- **Separate Generation**: Keep instances of `java.lang.Class` in a dedicated generation because they are generally long lived and contradict the *weak generational hypothesis*. This can be easily realized because those instances are exclusively instantiated by the *virtual machine* internally, as opposed to a `new` keyword. All references held in static fields are considered to be *old-to-young references* in the way that section 4.3 presented them. Collections in that generation are only initiated when class unloading is triggered by the following point and not when memory runs low.

- **Weak Class Cache**: The *class cache* mapping class loaders to actual class information they loaded, uses weak global references to class loaders. Thereby additional cleanup code that initiates class unloading can be associated with class loaders, without keeping them alive unnecessarily.

## 5.5 No Silver Bullet towards Garbage Collection

As a final statement we want to emphasize that there is *no silver bullet towards garbage collection*, especially when it comes to embedded systems or other resource-constrained environments [24]. Modern *garbage collection* implementations allow tweaking to fine-tune *garbage collector* behavior towards specific needs of a particular application, even production-ready Java solutions often are equipped with more than one *garbage collector*.

A lot of the general concepts presented throughout this thesis (like partitioning into regions, concurrent collections & thread locality) are essential for a state-of-the art *garbage collector* and need to be present. Their concrete configuration however, highly depends on the application being executed. As a result we believe that the goal should be to create a flexible *garbage collection framework* which can be tweaked to individual scenarios. Since manual tweaking is infeasible, profiling of the application's memory management needs and characteristics should be done and fed back into the framework to further improve performance. Just like *compiler profiling* can help to optimize the result of a compiler, *garbage collector profiling* could be used to optimize the results of a *garbage collector* [24].

Especially for embedded systems, making best use of the limited resources is a priority, regarding garbage collection as a separated black-box is not beneficial. Hence we see *garbage collection* as an integrated part of the *virtual machine* and believe it needs to be treated as such.

# Bibliography


[1] O. AGESEN, *GC points in a threaded environment*, tech. rep., SMLI TR-98-70. Sun Microsystems, December 1998.

[2] O. AGESEN AND D. DETLEFS, *Finding references in Java stacks*, in Workshop on Garbage Collection and Memory Management, OOPSLA '97, October 1997.

[3] O. AGESEN, D. DETLEFS, AND J. E. MOSS, *Garbage collection and local variable type-precision and liveness in Java virtual machines*, SIGPLAN Notices, 33 (1998), pp. 269–279.

[4] A. W. APPEL, *Garbage collection can be faster than stack allocation*, Information Processing Letters, 25 (1987), pp. 275–279.

[5] A. W. APPEL, *Simple generational garbage collection and fast allocation*, Software Practice and Experience, 19 (1989), pp. 171–183.

[6] D. E. BACON, S. J. FINK, AND D. GROVE, *Space- and time-efficient implementation of the Java object model*, in Proceedings of the Sixteenth European Conference on Object-Oriented Programming (ECOOP 2002), Springer, June 2002, pp. 111–132.

[7] H.-J. BOEHM, A. J. DEMERS, AND S. SHENKER, *Mostly parallel garbage collection*, in Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, PLDI '91, New York, NY, USA, 1991, ACM, pp. 157–164.

[8] H.-J. BOEHM AND M. WEISER, *Garbage collection in an uncooperative environment*, Software Practice and Experience, 18 (1988), pp. 807–820.

[9] C. J. CHENEY, *A nonrecursive list compacting algorithm*, Communications of the ACM, 13 (1970), pp. 677–678.

[10] D. DETLEFS, C. FLOOD, S. HELLER, AND T. PRINTEZIS, *Garbage-first garbage collection*, in Proceedings of the 4th international symposium on Memory management, ISMM '04, New York, NY, USA, 2004, ACM, pp. 37–48.

[11] 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*, Communications of the ACM, 21 (1978), pp. 966–975.

[12] T. DOMANI, G. GOLDSHTEIN, E. K. KOLODNER, E. LEWIS, E. PETRANK, AND D. SHEINWALD, *Thread-local heaps for Java*, SIGPLAN Notices, 38 (2002), pp. 76–87.

[13] D. GRUNWALD, B. ZORN, AND R. HENDERSON, *Improving the cache locality of memory allocation*, SIGPLAN Notices, 28 (1993), pp. 177–186.

[14] A. L. HOSKING, J. ELIOT, B. MOSS, AND D. STEFANOVIC, *A comparative performance evaluation of write barrier implementations*, in Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, October 1992, pp. 92–109.

[15] R. JONES AND R. D. LINS, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, Ltd., 1996.

[16] H. B. M. JONKERS, *A fast garbage compaction algorithm*, Information Processing Letters, 9 (1979), pp. 26–30.

[17] A. KRALL, *Efficient JavaVM just-in-time compilation*, in Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, PACT '98, Washington, DC, USA, 1998, IEEE Computer Society, pp. 205–212.

[18] A. KRALL AND R. GRAFL, *Cacao - a 64-bit JavaVM just-in-time compiler*, Concurrency and Computation: Practice and Experience, 9 (1997), pp. 1017–1030.

[19] S. LIANG, *Java Native Interface: Programmer's Guide and Specification*, Prentice Hall, 1999.

[20] T. LINDHOLM AND F. YELLIN, *Java Virtual Machine Specification*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd ed., 1999.

[21] P. MOLNAR, A. KRALL, AND F. BRANDNER, *Stack allocation of objects in the Cacao virtual machine*, in Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ '09, New York, NY, USA, 2009, ACM, pp. 153–161.

[22] F. L. MORRIS, *A time- and space-efficient garbage compaction algorithm*, Communications of the ACM, 21 (1978), pp. 662–665.

[23] M. PAWLAN, *Reference objects and garbage collection*, August 1998. `http://www.pawlan.com/monica/articles/refobjs/`.

[24] A. PETIT-BIANCO, *No silver bullet - garbage collection for Java in embedded systems*, August 1998. `http://gcc.gnu.org/java/papers/nosb.html`.

[25] C. PFEIFHOFER AND M. STARZINGER, *Garbage collection - BoehmGC*. June 2006.

[26] T. PRINTEZIS AND D. DETLEFS, *A generational mostly-concurrent garbage collector*, in Proceedings of the 2nd international symposium on Memory management, ISMM '00, New York, NY, USA, 2000, ACM, pp. 143–154.

[27] H. SCHORR AND W. M. WAITE, *An efficient machine-independent procedure for garbage collection in various list structures*, Communications of the ACM, 10 (1967), pp. 501–506.

[28] E. STEINER, A. KRALL, AND C. THALINGER, *Adaptive inlining and on-stack replacement in the Cacao virtual machine*, in Proceedings of the 5th international symposium on Principles and practice of programming in Java, PPPJ '07, New York, NY, USA, 2007, ACM, pp. 221–226.

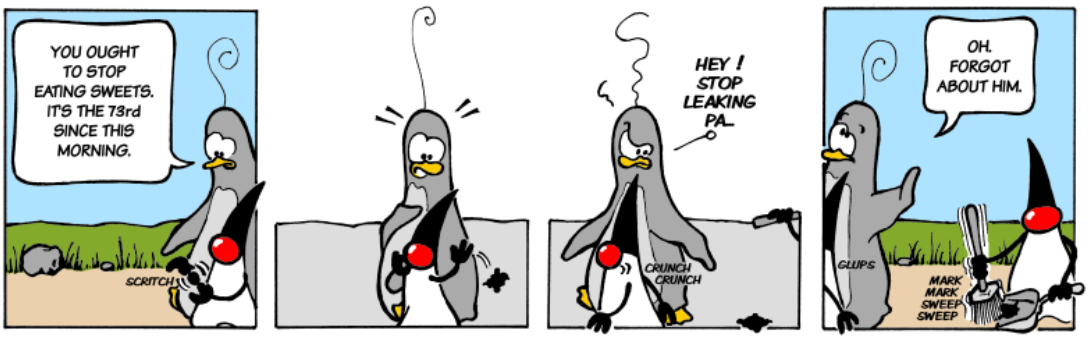
[29] D. UNGAR, *Generation scavenging: A non-disruptive high performance storage reclamation algorithm*, SIGSOFT Software Engineering Notes, 9 (1984), pp. 157–167.

[30] B. ZORN, *The measured cost of conservative garbage collection*, Software Practice and Experience, 23 (1993), pp. 733–756.

# List of Figures

# List of Listings






Comic courtesy of Laurent Grégoire.