Expiration Classes for Implicit Memory Management

Magisterarbeit

zur Erlangung des Diplomgrades
an der Naturwissenschaftlichen Fakultät der
Paris-Lodron Universität Salzburg

eingereicht von
Andreas Haas, Bakk techn.

Gutachter: Univ. Prof. Dr. Ing. Dipl. Inform. Christoph Kirsch

Fachbereich: Computer Science

Salzburg, August 2009

Danksagung

Den nachfolgenden Personen möchte ich an dieser Stelle danken:

Meinem Betreuer, Professor Christoph Kirsch, der immer Zeit für mich hatte und mich bestens unterstützt hat.

Ana Sokolova und Hannes Payer, mit denen ich viel über meine Arbeit diskutiert habe, und von denen ich viel gelernt habe.

Meiner Freundin Theresa und meinem Bruder Stefan, die meinem Englisch sehr auf die Sprünge geholfen haben.

Ich danke meinen Eltern, Bernhard und Michaela, die mich stets unterstützt haben.

Zuletzt möchte ich mich noch bei meinem Mitbewohner, Andreas Schönegger, bedanken, der mich während des ganzen Studiums begleitet hat und mit dem ich an vielen Projekten gearbeitet habe.

Abstract

Expiration classes are a concept to analyze memory management systems. It reveals the costs and the source of costs. Explicit memory management, garbage collection, region-based memory management and cyclic allocation are analyzed in this master thesis. Furthermore, we present AGC, a new memory management system which combines garbage collection with cyclic allocation. Cyclic allocation has very good timing properties and predictable space consumption, but it is hard to use. On the other hand, garbage collection is very convenient, but it has some unwanted side effects, like long pause times. AGC can use the advantage of both systems, with only small overhead. Benchmarks show that AGC is competitive to other garbage collected systems.

Contents

List of Figures

List of Algorithms

List of Tables

Part I

Introduction

1 Introduction

Memory management is one of the key aspects of every program execution. The first part of this master thesis deals with a new way to analyze memory management systems (MMS): Expiration classes. The expiration class analysis reveals the origin of costs in a conceptual way and makes them comparable.

Based on the results gained by expiration class analysis, we developed a new memory management system, called Annotated Garbage Collection (AGC). AGC is a combination of two other systems: tracing garbage collection and cyclic allocation.

In cyclic allocation, every allocation requires the same amount of time, which is very good. Moreover, its memory consumption can be calculated at compile-time. Nevertheless, sometimes it is difficult to use. On the other hand, garbage collection is very convenient because the programmer does not have to care about memory management. However, in nearly all implementations of garbage collection the allocation time is not predictable.

Cyclic allocation has predictable memory consumption and predictable allocation time. However, the programmer has to give additional information to use cyclic allocation. Thereby, she improves the predictability of the system. Therefore, the programmer should give as much information as possible, but she has to care about the correctness of this information. If it is not possible to provide correct information, garbage collection is used, which does not need additional information to be correct.

Benchmarks show that AGC is competitive to pure garbage collected systems.

When the programmer gives enough information, the number of garbage collection runs can be reduced significantly. Thereby, the program gets more predictable.

1.1 Outlining of this Thesis

The master thesis starts with an introduction of memory management. Thereafter, the concept of expiration classes is described. Fundamental approaches of memory management are analyzed in the view of expiration classes to determine their advantages and disadvantages. The following sections deal with AGC, a new memory management system we developed on the knowledge gained by expiration class analysis. The thesis ends with the results of benchmark tests we started to examine the properties of AGC.

Section 1: Introduction The introduction gives an overview of the master thesis.

Section 2: Memory Management This section describes the base concepts of memory management.

Section 3: System Properties The aim of this section is to introduce some properties which are important in memory management.

Section 4: Expiration Class This section deals with the concept of expiration classes. Some important memory management system are analyzed, like explicit memory management, garbage collection, region-based memory management and cyclic allocation.

Section 5: AGC: Conceptual Overview The key concepts of AGC are described.

Section 6: Other Hybrid Memory Management Systems As AGC is a hybrid memory management system, this section describes other hybrids and their solution to some key problems which arise if different systems are combined.

Section 7: Implementation Environment The system we extended with AGC is described in this section.

Section 8: AGC: Implementation This section deals with the changes done to an existing virtual machine.

Section 9: Experiments and Results We run several benchmarks with our system. The results are presented in this section.

Section 10: Conclusions This section concludes the thesis.

2 Memory Management

Every computer program needs memory to store its program code and data. The whole memory (called heap) is just one big space, and the program can write values everywhere into this space. However, for efficient use, this space is handled by memory management.

The program can request pieces of memory from the memory management when it needs it (this is called "to allocate memory"), and later, when there is no need for it anymore, the program gives this pieces back ("the memory is deallocated" or "the memory is freed"). It can be used again for other purposes. Allocated memory which may be deallocated at some future time is called dynamic memory. Otherwise, it is called permanent memory.

The main issue of memory management is to manage free memory. To allocate memory, the memory management has to find an appropriate portion of free memory. Deallocation implies that the deallocated memory is made suitable for later use. This can mean that the deallocated memory is added to a list of free memory. If the allocated memory is not contiguous, the memory management also has to help the program to find related data.

Modern, object-oriented programming languages like Java and C# do not allow the programmer to use raw memory directly. She can only work on objects.

Object An object is a piece of memory with a well-defined internal structure. The structure defines which positions in an object store which kind of information. For example, an object can contain four positions for numbers and three positions for characters. It is not possible for the programmer to violate the structure. For example, she cannot write a character at a number position. In object-oriented programming languages, only objects can be allocated, not raw memory. The allocated memory is called the memory of the object.

The internal structure of an object is defined by its object type.

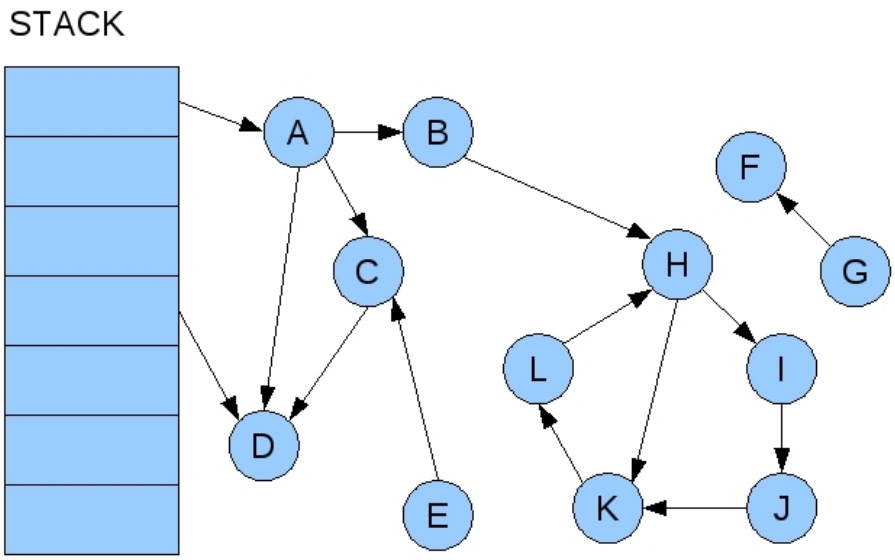


Figure 1: Object graph

Object type An object type defines the internal structure of an object. Every object has exactly one object type, but many objects can have the same object type.

The program uses special data to find objects in the heap. This data is called references.

Reference A reference, also called a pointer, is a kind of data which contains position information about other data. This other data can again be an object. In objects, some positions can contain references. Thereby, objects can refer to other objects.

Object graph As objects can contain references to other objects, the program can create an object graph, where objects are vertices, and references are directed edges.

The program needs some entry points to access the object graph. This entry points are given by data which is stored at special positions in memory. Two kinds of data can specify entry points: Global variables and the stack. The stack stores temporary data whereas global variables often store permanent

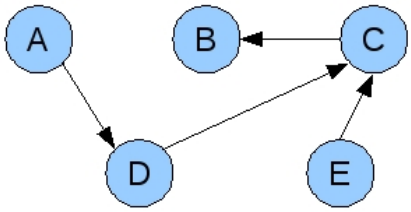


Figure 2: Example of an object graph.

data. Global variables can be simulated by the stack. Therefore, for the rest of the thesis we drop the notion of global variables. The entry points of the object graph are only specified by the stack. Figure 1 presents an example of an object graph. Object A and Object D are the entry points.

To find an object, the program has to follow a path in the object graph, starting at the stack and leading to the object. The object graph is very important for some memory management systems. Therefore, we define some notions which are required later.

Reachability Object A can reach object B if there is a path in the object graph starting at A and ending in B. In this case A is a start object and B is reachable. An object can also be reachable if there is a sequence of references starting at the stack. In Figure 2, object A can reach object B, because there exists a sequence $A \rightarrow D \rightarrow C \rightarrow B$. Object A cannot reach object E.

Liveness An object is called "live" if it is reachable from the stack. Otherwise it is called a "dead" object. In Figure 1, object E, F and G are dead because they are not reachable from the stack. All other objects are live.

Liveness is a very important property in memory management. Objects which are not live can be deallocated because the program cannot use them anymore. The program simply cannot find the position where the object is stored.

Live Time The live time of an object is the time interval in which the object

is live.

A program always has to tell the memory management when it needs additional memory. However, memory deallocation can be done by the memory management itself. It just has to check if an object is dead, because dead objects can be deallocated. These objects will not be used anymore. This type of memory management is called "implicit memory management".

Implicit memory management systems are very convenient for programmers. It is much easier to determine if additional memory is needed than to know that some pieces of memory are not needed anymore. Lots of software errors would not exist if implicit memory management would be used. However, this comfort is not for free. It is not easy to determine the liveness of all objects. The timing of programs using implicit memory management is unpredictable, which is very bad for time critical applications, like control software in cars.

Memory management which requires deallocation information of the program is called explicit memory management. Explicit memory management is faster than implicit memory management, but it is not easy to use explicit memory management correctly. Due to improper use, memory violations can happen.

Memory violation Memory violations happen if the program stores data at positions which are already used elsewhere.

Memory violations are errors which are hard to find. Computer viruses can use memory violations to infect a computer, because they overwrite important parts of the memory of a program.

Correctness A memory management system which does not allow memory violations is called correct.

Most implicit memory management systems are correct. However, if a system is not correct, it can be at least type-safe.

Type-safe A system is called type-safe if the data of an object can only be used according to its intention. For example, numbers cannot be used

as references, and references are not used for calculations.

A type-safe system need not be correct, nevertheless, some kinds of program malfunction cannot happen. For example, if a number is used as a reference, it can point at a position in memory where program code is stored. If the program stores data at this position, the program code gets corrupted. In a type-safe system, this scenario cannot happen because a number cannot be used as a reference.

3 System Properties

This section deals with some properties a system can have. First we will describe time dependent properties. Thereafter, we will introduce some memory dependent properties.

3.1 Time specific properties

The time a computer program needs for its calculations is very important. It is annoying to wait for the computer to complete its work. On the other hand, what is a very fast program for, if it freezes every now and then for some seconds and does not react to any input. These properties are described in this section.

The first property we want to describe is the "total runtime".

Total Runtime The "total runtime" of a program or a part of a program is the time that passes from the start of the part of program until its end.

The total runtime is very important for calculations, but not very important for programs with user input. The fastest program seems slow if it waits for user input most of the time. For this kind of programs, the runtime between user input is important.

The notion of "runtime" does not only denote the time span a program needs for its execution. It is generally used to speak about the time the program is running. For example, "Something is done during program runtime" means that something is done in the time the program is executing.

A related notion of "runtime" is "compile-time". For example, some system checks can be done at compile-time. This means that the checks can be done before the program is actually executed. This is very good, because compile-time checks only have to be done once, and they do not influence the timing during program execution.

As we already mentioned in the introduction of this section, it is rather annoying if an application freezes for some seconds. This can become catastrophic if it happens in a brake controller of a car. The time a program does not show any progress is called pause time.

Pause Time In the context of this master thesis, the pause time is defined as the maximum time it takes for a single memory management action, like allocation and deallocation, to execute. It is the maximal time a program may have to wait without making progress due to memory management activities.

For example, the allocation of memory is a source of pause times. Some memory management systems are very fast for most allocations, but every now and then, additional calculations are necessary, which let the program freeze for seconds. However, this does not mean that the memory management system is slow according to the total runtime. It can be even faster than a system without pause times.

Total runtime and pause time also exist in other systems like communication networks, but they have other names: Throughput and latency. Throughput is the amount of data delivered in a specific time span. Latency is the time needed to deliver one datum. Throughput versus latency is a fundamental trade off. In lots of systems, it is possible to process a set of tasks faster than every task one by one.

Properties or actions can depend on various factors. For example, sorting a list depends on the size of the list. If the list gets longer, sorting takes more time. Another example is the allocation time of a memory management system.

Allocation Time The allocation time is the time needed to allocate a single object. The allocation time can depend on the size of the object, or on the size of the heap, or on number of objects already allocated. The allocation time is called "predictable" if it only depends on the size of the allocated object.

Deallocation Time The deallocation time is the time needed to deallocate a single object and to make its memory suitable to be reused. Sometimes it pays off to reorganize the free memory at every deallocation to allow faster allocation afterwards.

3.2 Space specific properties

The memory available for a program is finite. When all memory is used, every following allocation will fail. The most obvious space specific property is the total memory consumption.

Total Memory Consumption The total memory consumption is the amount of memory needed to provide every allocation the requested amount of unused memory. Total memory consumption is also called "maximum heap-size".

This property is unknown for most of the programs, because everyone is happy as long as the program runs, and if not, the users have to buy additional memory. However, for systems with limited resources, this property is very important.

If a program does not deallocate its unused memory, the memory consumption increases until perhaps the whole memory gets full. This is called a "memory leak".

For some implicit memory management systems, the maximum heap-size is irrelevant because it is configurable. As long as the set heap-size is big enough, the system will be executed. For these kind of systems, the live-heap size is more important.

Live-Heap Size The live-heap size is the maximum amount of live memory in a program.

If a perfect explicit memory management would replace the implicit memory management, the live-heap size would be the same as the maximum heap-

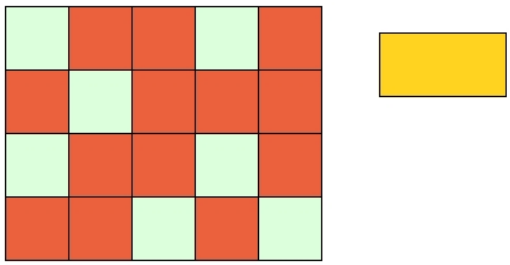


Figure 3: Example for fragmentation

size. It is good to know the live-heap size because the heap-size of an implicit memory management system has to be set when the program is started. If the configured heap-size is smaller than the live-heap size, the program will run out of memory.

The maximum heap-size is predictable if it only depends on the program and not on the time the program is running. This is very important for long running programs like web servers. A real-time memory management system requires both allocation time predictability and heap-size predictability.

Another important memory property is fragmentation.

Fragmentation A system suffers from fragmentation if the free memory of the system is not contiguous. Fragmentation, sometimes also called external fragmentation, is a property of the system, not of the program. As fragmentation influences the maximum heap-size, the heap-size of the program depends on the system.

Figure 3 shows an example for fragmentation: The rectangle to the left represents the memory. The red inner squares are allocated objects. Green represents free parts of the memory. As the green squares are not contiguous, the system suffers from fragmentation. The system wants to allocate the new yellow object of size two. This is not possible, although seven memory fields are unused. No piece of free memory is big enough to store the object.

There are some interesting things about fragmentation:

- Fragmentation may increase the allocation time. In the example of Figure 3, the memory management system has to scan over all pieces of free memory to find out that the yellow object cannot be allocated.
- Fragmentation can be eliminated by moving used memory together. This is called compaction. Compaction is difficult because references all over the heap may have to be changed if their target objects have moved.
- If all allocated objects have the same size, then fragmentation may have no consequences.

Part II

Concepts

4 Expiration Class

This section deals with the concept of expiration classes. There are lots of different memory management systems available, which are hard to compare. There is the concept of explicit memory management using explicit free()-calls for deallocation of every object. Then there is the concept of region-based memory management, which allocates objects that belong together into the same memory region. A third fundamental approach is the use of garbage collection, which deallocates memory automatically when it is no longer reachable.

For each of these methods there exist a lot of different implementations. It is very hard to compare them, as they all are specialized for different parts of memory management. The expiration class analysis allows to compare different memory management systems in a conceptual way.

First of all, some definitions are required:

Expiration Class An expiration class is a set of objects which are deallocated at the same time.

Every memory management system uses expiration classes. Otherwise, no memory can be used again. Therefore, expiration classes are fundamental for every possible implementation. There are some questions, which are important to analyze a memory management system:

- How is an expiration class managed?
  - If every expiration class contains just a single object, expiration class management is very simple.

  - When it contains more than one object, one way to manage them is to store them in a contiguous piece of memory.
  - Objects in an expiration class can be organized by an additional data structure.
  - It is possible that there is no management at all, but then, the objects of the expiration class have to be determined in an additional step before the expiration class is deallocated.

- How big are expiration classes?
  - If an expiration class only contains one object, fragmentation is very likely because objects have arbitrary size.
    - In [12], an implementation of a memory management which uses one-object expiration classes and which provides predictable fragmentation is presented. Only if external fragmentation is predictable, memory consumption and allocation time can get predictable too.
  - Bigger expiration classes can consist of memory blocks of the same size. Fragmentation has no consequences then.
  - If an expiration class is larger, but its objects do not get dead at the same time, one talks about internal fragmentation.
  - Every expiration class might contains some additional management data.

- How fast can all objects in an expiration class be deallocated?
  - For the runtime of a program, deallocation time is as important as allocation time. With a clever design, the deallocation time of an expiration class is always the same, independent of the size of the expiration class. When there are many object in an expiration class, the deallocation time per object gets very low. The question

is, which objects should be put into the same expiration class, and when can it be deallocated.

- What kind of objects does the expiration class contain?
  - An expiration class can contain objects which will all get dead at a similar time. Furthermore, it is possible that all objects in an expiration class must have the same size or the same object type.
- How does an expiration class affect other expiration classes?
  - In some memory management systems, the expiration classes depend on each other. Therefore, one expiration class cannot be deallocated before another class has been deallocated. This relationships have to be managed, too.
- Who decides which objects belong to the same expiration class?
  - Some memory management systems allow the programmer to group objects together, other systems decide it themselves.
- When can the objects in the expiration class be deallocated?

To answer the last question, another definition is required.

Expiration Event The event which leads to the deallocation of an expiration class is called expiration event.

Expiration events are as fundamental in memory management systems as expiration classes. The time when an object can be deallocated is very important, as wrong timing leads to an incorrect system. Events are the most abstract way to define time.

An expiration event can be anything that happens during program execution. For example, the execution of a special command is an event. The question is how to detect the expiration event. The answer of this question leads to the basic properties of the memory management.

There are some aspects one has to keep in mind when looking for expiration

events. For every memory management system, there exists a time interval in which a correct expiration event has to occur. The interval starts at the last time an object of the expiration class has been accessed, and it ends when the expiration class cannot be found anymore. Invalid expiration events lead to two serious kinds of malfunction:

Dangling pointers Dangling pointers occur when an object is deallocated, but there still exist some references to it. When the memory of the object is used again, its content is invalid for these references. By reading corrupt data via such references, the program may behave unexpectedly. Dangling pointers are a common source of memory violations.

Memory leaks Memory leaks occur when the programmer does not deallocate memory at all, or she does it to late. If the program is running very long, and it creates memory leaks all the time, it may run out of memory and be unable to continue.

The Dangling pointer problem happens when the expiration event is triggered too early. Memory leaks arise when the expiration event does not occur at all. Both indicate an incorrect and unpredictable computer program.

In the following sections, we will analyze some memory management systems using expiration classes, and we will show that the approach is meaningful. The systems are ordered by their expiration event and the size of their expiration classes.

4.1 Explicit Memory Management

Explicit memory management is one in which the programmer takes care of allocating and deallocating objects. In view of expiration class analysis, this denotes that she has to trigger the expiration events herself. The memory management system cannot force the programmer to use the expiration events in a correct way. Therefore, both dangling pointers and memory leaks are possible. The problem is that the programmer cannot wait until an ob-

ject is dead to deallocate it, because then, she has no information to tell the memory management system which object it should deallocate. This is the origin of the dangling pointer problem. An object should be deallocated when there is exactly one reference left to reach it.

All in all, the programmer has the responsibility to use the memory management correctly. The memory management only provides the functionality of allocating and deallocating memory.

The next two sections present such explicit memory management systems, which only differ in the size of their expiration classes.

4.1.1 Explicit Memory Management using a simple free()-statement

One of the most used memory management system is an explicit memory management system using expiration classes which contain just one object. The programmer has to set a free()-statement in the program code whenever she wants to deallocate an object.

This kind of memory management system has several disadvantages, which we already discovered in the previous sections, such as

- invalid expiration events: sometimes it is difficult to set free()-statements correctly. If the programmer triggers expiration events outside the correct time interval, both dangling pointers and memory leaks may occur.
- fragmentation: As every object can have arbitrary size, this system has to care about fragmentation because it can influence allocation time and memory consumption.

4.1.2 Region-based Explicit Memory Management

Region-based memory management systems are systems which have well-defined expiration classes, called regions, containing one or more objects.

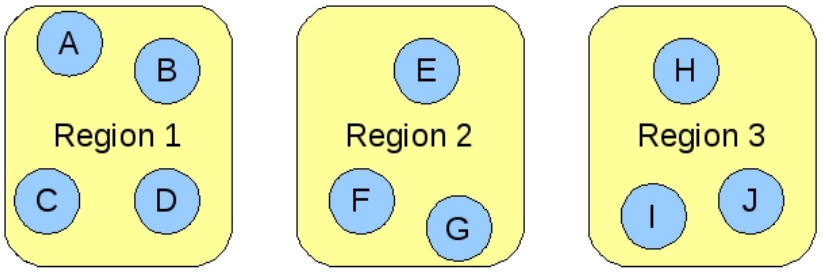


Figure 4: Region-based memory management

This means that at every moment, an arbitrary expiration class knows which objects it contains. For example, in Figure 4, three different regions contain ten objects.

There exist many implementations of region-based memory management systems, which differ in their handling of expiration classes and in the detection of expiration events. In explicit region-based memory management, the programmer decides which objects belong together to the same expiration class, and she triggers the expiration event herself.

The used expiration classes and expiration events lead to the following properties:

- As every explicit system, dangling pointers and memory leaks can occur. In this system, it is even more likely, because if the programmer assigns an object to the wrong expiration class, it will be deallocated too early or too late.
- Low deallocation time per object. As stated before, it is possible to organize an expiration class in a way that it can be deallocated independent of its size. Thereby, the deallocation time per object gets low.
- Fragmentation can be avoided. Nevertheless, the expiration classes themselves suffer from internal fragmentation. However, the programmer has to care about it. If the programmer can estimate the internal fragmentation, the maximum heap size can be estimated. Moreover,

allocation time and deallocation time can be predictable too. Internal fragmentation neither influences the allocation time nor the deallocation time.

Because of the third property region-based memory management systems are used a lot for real-time programs. Neither the maximum heap size, nor allocation time nor deallocation time depends on the memory management system. All three properties can be estimated by only knowing the program. Nevertheless, it is still difficult to estimate these three properties.

4.2 Implicit Memory Management: Garbage Collection

Garbage collection is one kind of implicit memory management, which means that the programmer has to allocate memory when she needs it, but she does not have to deallocate it. Deallocation is done by a garbage collector. It is very convenient for the programmer to use garbage collectors because she does not have to care about the live time of the used objects. The garbage collector determines during runtime if an object is live. If it is dead, the object is deallocated.

In the following sections, we will present some kinds of garbage collection. First we will analyze tracing garbage collection and reference counting garbage collection, which are the basic solutions for a garbage collected system. Thereafter, we will introduce some combinations. What we will see is that either the management of expiration classes is expensive, or the detection of the expiration events is.

4.2.1 Tracing Garbage Collector

A tracing garbage collector was first described in [19]. When there is no memory left on the heap, the garbage collector stops the program. Then it searches through the stack for references to the heap, called root references.

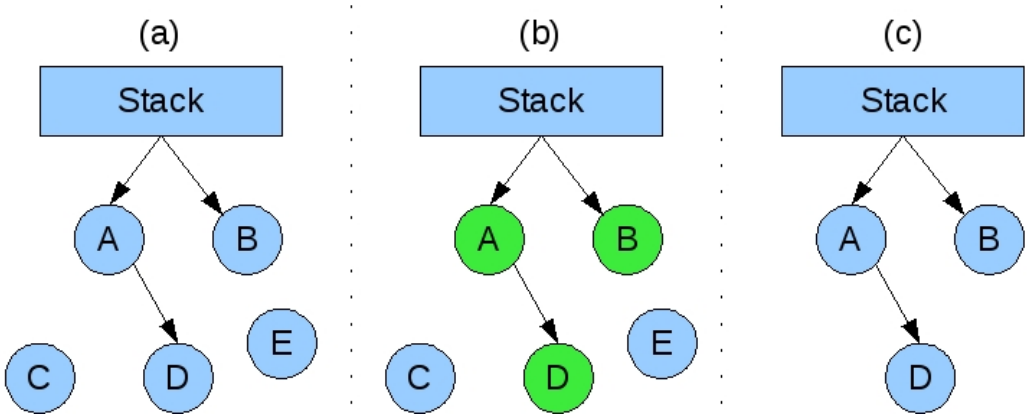


Figure 5: Example for tracing

Root An object on the heap is called "root" if a reference on the stack points to it. Such a reference is called "root reference".

Starting from the root references, the collector iterates through the object graph, always following all references it can find. The collector marks each object it visits with a flag. This is called the "marking phase". The objects which are marked are called reachable.

After the marking phase the garbage collector sweeps over the heap and deallocates every object which is not marked. This is called the "sweeping phase".

An example for tracing can be seen in Figure 5. Figure 5(a) shows the object graph before the garbage collector starts. In Figure 5(b), the marking phase already passed. Object A and B are marked because root references point to them. Object D is marked because the marked object A contains a reference to it. The objects C and E are not marked, because they are not reachable. They are deallocated during the sweeping phase. Figure 5(c) shows the object graph after garbage collection.

The expiration event therefore is triggered when the heap is full and no additional memory can be allocated. This is not expensive to track and most memory management systems have to check it, because they have to check

if there is free memory left to allocate.

Nevertheless, expiration classes are hard to determine. The whole marking phase only exists to determine an expiration class. This has the advantage that between two collection runs the memory management does not do anything. A system using a tracing garbage collector is unbeatable in performance between two collection runs.

Anyway, tracing still has all the overhead other systems have distributed over the whole runtime: a collection run is pure overhead. Only the system benefits of it, not the program. An implicit system promises to provide memory, and it has to collect dead objects to keep this promise.

The determination of expiration classes leads to long pause times, because the program cannot make any progress during a collection run. This could violate the correctness of the system. Moreover, the pause time depends on the size of the heap. When the heap-size increases, the pause times get longer. This can be rather annoying for the user. In safety-critical applications, this is unacceptable.

Tracing garbage collection has a second disadvantage as well. The workload of the garbage collector does not depend on the number of unreachable objects, but rather on the number of live objects and the size of the whole heap. Therefore, when the program has lots of permanent data, the garbage collector always runs a long time, but deallocates few objects.

In summary, the lack of expiration class management leads to fast execution, but also to long pause times, which can be very annoying.

4.2.2 Reference Counting Garbage Collector

Reference counting is a second fundamental way to do garbage collection. It was introduced in [11]. The garbage collector does not stop the program and therefore it has no problem with pause times. This is possible by using reference counters on every object. A reference counter stores the number

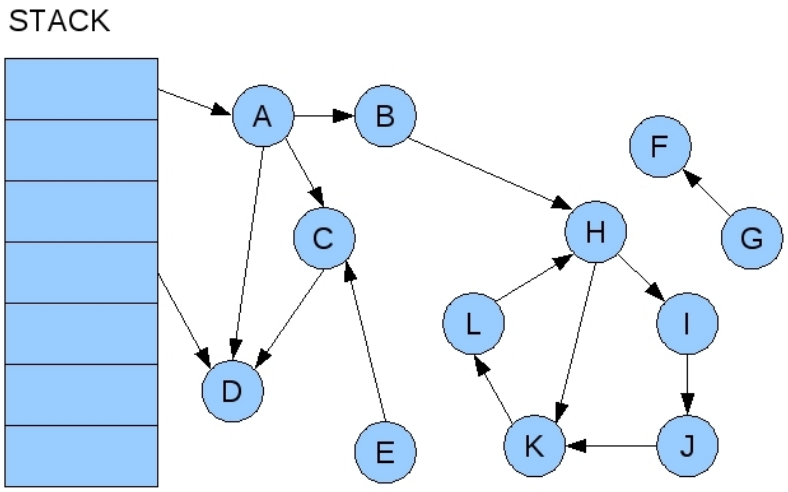


Figure 6: Object graph with cycles

of references which point to an object. When a reference is changed, the counter of the previous target is decremented and the counter of the new object is incremented. When a reference counter gets zero, the object can be deallocated. All references of this object are deleted before deallocation, which leads to a change of other reference counters.

For example, in Figure 6, the reference counter of object D is three, because three references point to it. No reference points to object G, therefore it can be deallocated. When this is done, the reference to object F is deleted, so object F can be deallocated too.

The expiration classes of this system only consist of one object. As for the other systems with this kind of expiration classes, this leads to the problem of fragmentation.

Moreover, it is expensive to detect the expiration events. Programs update references very often. Therefore, this operation should be very fast. As one can see in Figure 7, keeping the reference counters up to date leads to four memory operations instead of just one. Memory operations are some of the most expensive operations a program can execute. Instead of just writing the new reference value, the reference counters of two additional objects have to

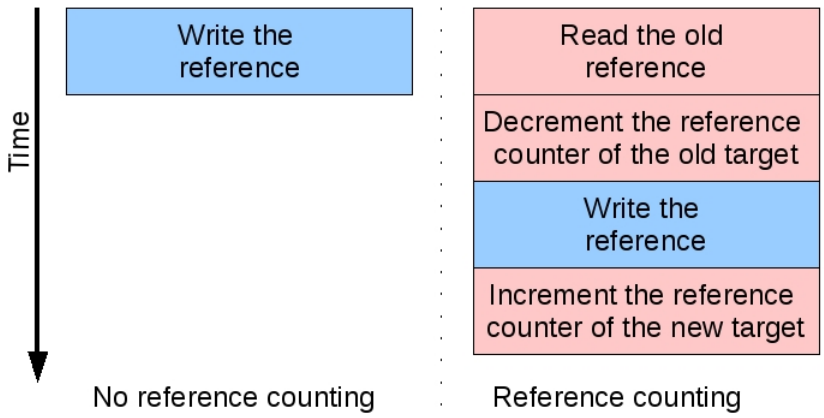


Figure 7: Reference counting overhead when a particular reference is changed

be changed too. In most cases, a tracing garbage collector beats a reference counted system in performance, because the determination of the expiration classes in a tracing system is cheaper than the detection of the expiration events in a reference counted system.

Reference counters are a way to approximate the liveness property of objects. However, this approximation is not exact because there may exist cycles in the object graph.

Cycle A cycle is a set of objects with the property that for any two objects A and B in the cycle, A can reach B. Therefore, any object in a cycle can reach itself. In Figure 6, an example of a cycle is formed by the objects H, I, J, K, and L.

When there are cycles, the objects in any cycle are live, because the references within the cycle are counted as well. Even if all objects of the structure are unreachable by the program, the reference counter does not get zero, and hence the expiration event is not triggered. These objects have to be deallocated by a fallback mechanism.

As illustration, in Figure 6, the cycle {H, I, J, K, L} is only reachable from object B. When this reference gets deleted, the reference counter of H remains one because of the reference coming from L. This reference will never

be deleted, because the Object L is unreachable and therefore cannot be changed.

4.2.3 Garbage Collection in general

As stated in [7], every garbage collector is a combination of tracing and reference counting. With some adaptions, the marking phase of a tracing system can calculate exact reference counters, which even ignore cycles. This can improve reference counting, because it allows to ignore some changes of reference counters. If some reference counters are zero, a marking phase is started to calculate the exact reference counters of this objects. If a reference counter is still zero, the object can be deallocated.

On the other hand, additional reference counting allows a tracing system to process distinct parts of the heap separately. Thereby, the time of a collection gets independent of the heap size, it depends only on the size of the heap segment.

In view of expiration classes, in a hybrid system of reference counting and tracing, expiration classes are determined faster than in tracing, but slower than in reference counting, whereas the detection of expiration events is faster than in reference counting, but slower than in tracing.

To analyze an arbitrary garbage collector, one has to find out what kind of heap segments are used, how these segments are handled, and how references between heap segments are handled. The stack can be seen as a special heap segment, which need not be collected but which influences the live times of objects in other segments.

An example is a two layer generational garbage collector [5, 22], as illustrated in Figure 8. The heap is split into two parts, called mature space and nursery. All objects are allocated into the nursery. When it is full, garbage collection starts. After the nursery is collected, all live objects are copied from the nursery to the mature space. Therefore, when the nursery is collected again,

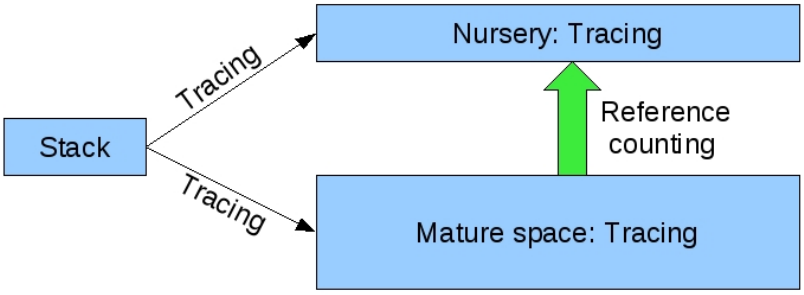


Figure 8: A two layer generational garbage collector.

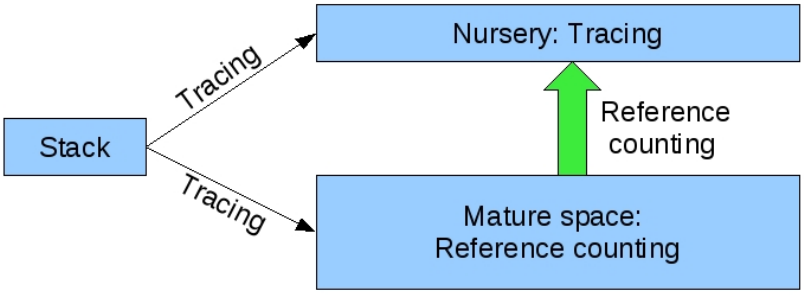


Figure 9: Ulterior reference counting

these objects are not considered, which reduces the time for both phases of Tracing. References from the mature space to the nursery have to be processed too, because they can keep objects in the nursery live. These references are handled by reference counting.

The mature space is only collected if the nursery has been collected before. Therefore, during the marking phase of the mature space, the nursery is empty, and no references from the nursery to the mature space can exist.

To sum up, the contribution of a generational garbage collector is to speed up the determination of expiration classes by reducing the search space. However, the detection of expiration events gets more expensive.

Another example of a hybrid garbage collector is ulterior reference counting, described in [10]. Ulterior reference counting is a generational garbage collector too, but it manages the mature space with reference counting. It has

been observed that references in the mature space do not change very often and therefore detection of expiration events is not that expensive and pause times are reduced, too. References from the stack are ignored, because they change a lot. When the reference counter of an object gets zero, an additional tracing phase checks whether the object is reachable from the stack. Ulterior reference counting is shown in Figure 9.

It is very hard to make the allocation time of a garbage collected system predictable. It would be possible to implement a predictable reference counting garbage collector, which implies predictable allocation time and predictable memory consumption. However, reference counting is not able to collect cycles. A tracing garbage collector collects cycles, but allocation time is not predictable because of long pause times.

One way to use real-time programs with garbage collection is to create a collector which can process small, independent pieces of memory and the time needed for one piece is known. The collector works in parallel to the program all the time. If the program does not need more memory than the collector can deallocate in the same time, it is possible for the program to use this memory management system. One such garbage collector is described in [6].

4.3 Implicit Region-based Memory Management

In section 4.1.2, we already introduced region-based memory management systems. We stated that region-based memory management systems differ in their handling of expiration classes and in the detection of expiration events.

The definition of expiration classes as well as the definition of expiration events can be done by the programmer, it can be done by program analysis at compile time, or it can be done at runtime. Program analysis is difficult, the programmer makes mistakes, and doing it at runtime is expensive. Anyway, all combinations lead to a new kind of memory management. As an example, we will analyze just one method of implicit region-based memory

management, which is presented in [21].

The method presented in [21] defines expiration classes as follows: Objects which are connected belong to the same expiration class. Two objects are connected if the first object has a reference to the second, or the other way around. Therefore, two objects from different regions cannot be connected. The mapping of objects to regions can be done at compile time by program analysis, which minimizes management costs at runtime.

The expiration event is triggered when the last reference from the stack to an object in a region is destroyed. The paper does not describe how this is done. We assume that some kind of reference counting is used. The expiration event would be detected at runtime then. As only references are considered which are not stored in objects, no cycles are possible. Therefore, a problem of reference counting, cycle detection, is avoided.

The determination of expiration classes is the main problem of region-based memory management systems. For example, in the presented method, one permanent object keeps a whole region alive, even if it contains lots of short living objects. It is even possible that there only exists one expiration class for the whole program. In this case, no memory will ever be deallocated. This is called the region-explosion syndrome. The region-explosion syndrome is an extreme form of internal fragmentation. Region explosion also occurs in most other region-based systems.

4.4 Cyclic Memory Allocation

Cyclic memory allocation is described in [20]. It is a very special way of memory management. It uses one-object expiration classes, but it does not suffer from fragmentation. It does not detect expiration events at all, and it does not really deallocate objects.

To understand the system, we first have to introduce some notions.

Allocation site An allocation site is a command in the program code which

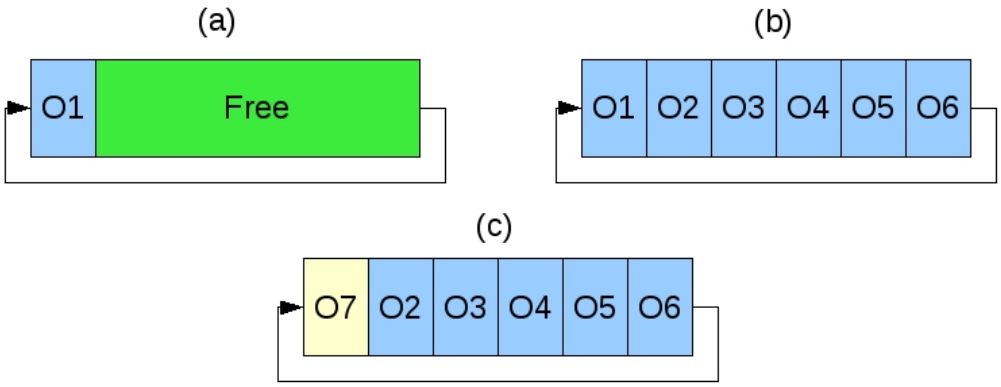


Figure 10: Cyclic buffer with size 6

allocates new memory. For example, in the programming language C, every malloc()-statement is an allocation site.

Furthermore, the concept of cyclic buffers is important:

Cyclic buffer A cyclic buffer is a special implementation of a buffer. When the buffer is full, the next allocation will overwrite the first object in the buffer. The buffer has a fixed size. One has to choose the size of the buffer carefully. If it is too small objects are overwritten too early. A cyclic buffer is shown in Figure 10. The first object O1 is allocated at the beginning of the buffer. The following five objects are allocated exactly after the previous object. When the program wants to allocate O7, there is no memory left at the end of the buffer. Thus, the memory of object O1 is reused.

The intention of [20] is to eliminate memory leaks. This memory management system is called cyclic memory allocation, because it uses a cyclic buffer for every allocation site. Thereby it restricts the amount of memory which can be allocated at a specific allocation site. If the cyclic buffer of an allocation site is full, the next allocation at this site will overwrite the first allocated object. Thereby, the memory required by a specific allocation site is fixed.

The size of the buffers is important. It has to provide enough memory for

all objects which can be live at the same time. Otherwise, live objects will be overwritten. Nevertheless, even if the buffer is too small, the system is at least type-safe, because the overwritten object and the newly allocated object have the same object type. Therefore, memory violations cannot happen.

For the expiration class analysis, cyclic memory allocation is special because no explicit deallocation is done. The memory of an object is just reused, it is not deallocated before. Therefore, the expiration event is triggered implicitly when an object overwrites another one. This event does not have to be detected.

The expiration classes only contain one object, nevertheless, the system does not suffer from fragmentation, because the objects in a buffer all have the same size. Only internal fragmentation can happen if some buffers are too big. The total memory consumption is predictable. It is the amount of memory required for the cyclic buffers.

The allocation time is predictable too. Every allocation site only needs to know the position of its buffer. The cyclic buffers are not moved during program runtime. If an allocation site remembers the position of its buffer once, it can find the buffer instantly at all following allocations. Buffer handling itself does not depend on anything except the buffer. Therefore, allocation time is predictable.

The problem is that it is not possible to define proper buffer sizes for all allocation sites. In [20], the authors tried to determine suitable buffer size using test runs. However, when the number of live objects in a buffer depends on the input of the program, it is likely to set insufficient buffers.

4.5 Hybrid Memory Management Systems

All presented memory management systems have advantages and disadvantages, and people tried to combine them. The costs of the resulting system are a combination of the costs of the subsystems. Moreover, the combination

itself implies some additional costs.

The hybrid systems do not introduce new expiration events nor new ways of expiration class handling. Nevertheless, the way how to combine different systems is interesting. We developed a hybrid system too, and we will present our solution in the next section.

By presenting our system, we will expose some common problems of hybrid systems. In Section 6 we will then present the solutions of other combined systems.

4.6 Summary

We analyzed the most important implementations of memory management systems. We saw that most systems use easy-to-detect expiration events, because detecting expiration events can be expensive, as it is the case for reference counting. Systems with common expiration events share common properties.

The handling of expiration classes is important for space-specific system properties. Lots of systems which use one-object expiration classes suffer from fragmentation. On the other hand, sometimes it is difficult to properly define bigger expiration classes. If the expiration classes are determined at runtime, long pause times can occur.

A short overview over the presented memory management systems can be found in Table 1.

System	Expiration Class	Expiration Event
Simple explicit MMS	One object	free()-statement
Region-based explicit MMS	Programmer-defined region	freeRegion()-statement
Tracing garbage collector	All dead objects found in the marking phase	Heap is full
Reference counting	One object	Reference counter gets zero
Region-based implicit MMS	System-defined region	Region is unreachable
Cyclic memory allocation	One object	The object gets overwritten

Table 1: Memory management systems, expiration classes and events

5 AGC: Conceptual Overview

This section introduces Annotated Garbage Collection (AGC), a memory management we developed on the insight we got in the last section. We wanted to create a memory management system satisfying the following properties:

1. It should be applicable for the programming language Java. Java programs typically run in a virtual machine, which uses any kind of garbage collection for memory management. Therefore, the language only supports allocation. No constructs for deallocation are available.
2. The programmer should be able to improve the behavior of the system by giving additional information.
3. When the programmer gives enough information, allocation time and memory consumption should get predictable. These two properties are very important for real-time programs, which absolutely require correct timing.
4. The performance of our system should be competitive to other systems.
5. It should be easy for the programmer to write a correct program.

5.1 Expiration Classes and Expiration Events

As we have seen in the previous section, it is important to have expiration events which are easy to detect and manage. It is not necessary that all expiration classes depend on the same kind of events. Therefore, we use a combination of events, which are all easy to handle, namely we combine the event for a tracing garbage collector, "the heap is full", and the implicit event of cyclic memory allocation.

Cyclic memory management satisfies all except the first and the last property. It is difficult to apply it on java programs, and sometimes it is hard to define

the correct buffer sizes. A garbage collector is always convenient to use and it is correct, but it fails on Property 2 and 3. In combination, one starts at the correctness of garbage collection and improves the other properties by using cyclic memory allocation as far as correctness is not violated.

The performance of the resulting system will be good. The costs of detecting the first expiration event are insignificant, as every memory management has to detect if the heap is full. Otherwise, it could not grant a consistent memory, because no additional objects can be allocated in a full heap independently of the used memory management system. As we saw in Section 4.4, the event of cyclic memory allocation needs not be detected at all.

When the heap is full we have to determine the expiration class of the tracing garbage collector. This is done by the marking phase. The expiration classes of cyclic allocation needs no special treatment, because they are deallocated implicitly.

If collection runs are neglected, a tracing garbage collector is very fast. We reduce the number of collection runs by reducing the number of objects which are managed by the garbage collector. Therefore, it is not absolutely necessary to use cyclic allocation for all objects. Yet it is important to keep in mind that allocation time is only predictable if no collection runs are necessary.

5.2 Memory Partitioning

To use both memory management systems, we split the heap into two parts: The first segment is managed by a tracing garbage collector (called GC heap), the second by cyclic memory allocation (called cyclic heap). However, there are some problems: The garbage collector should only consider objects in its part of the heap, since otherwise, we could not reduce pause times and surpass simple garbage collection.

Still we have to consider references from the cyclic heap into the GC heap

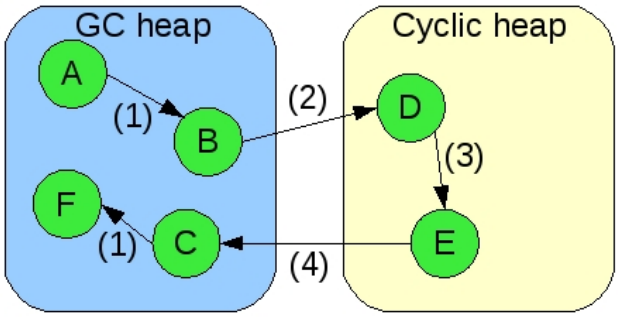


Figure 11: Possible references in AGC

during the marking phase, because maybe some objects in the GC heap are only reachable from objects in the cyclic heap. Another problem is that invalid references might exist if some buffers are too small, and a garbage collector may gets incorrect if it processes invalid references. Every memory management which uses a combination of garbage collection and a second system has to include solutions for these problems. We will present our solution in the subsequent sections.

5.3 Reference Handling

In this section, we will describe the management of all kinds of references. There are four different kinds of references in our system, as it can be seen in Figure 11:

1. References from the GC heap to the GC heap. These are the only references which have to be considered by the garbage collection in the marking phase. All these references are valid. The source object as well as the target object of the reference are managed by the garbage collected system, which does not allow to create invalid references. This kind of references is handled by the garbage collector, so no additional treatment is required.

2. References from the GC heap to the cyclic heap. They do not matter at all. The expiration events of objects in the cyclic heap do not depend

on references. These references can also be invalid. The target object might already be deallocated.

3. References from the cyclic heap to the cyclic heap. They do not influence the system either for the same reasons as references of type 2.

4. References from the cyclic heap to the GC heap. These are more difficult to handle. They have to be considered in the marking phase of the garbage collector, because it may be that some objects in the GC heap are only reachable through the cyclic heap. For example, in Figure 11, object C is live, because it can be reached from Object E. However, when only references of GC heap objects are considered, object C would be deallocated.

To solve the problem of the references of type 4, we apply reference counting, as it is done for generational garbage collectors. Every object in the GC heap has a reference counter for references coming from the cyclic heap. In Figure 11, the reference counter of object C has the value one, all other counters have value zero.

It is not enough to stop the deallocation of objects with reference counter greater than zero. Object F in Figure 11 would be deallocated then. The marking phase would ignore Object C, because it is not reachable from objects in the GC heap. It is not deallocated since its reference counter is one. However, object F is only reachable from C, and if C is not handled during the marking phase, F is not treated either. Moreover, the reference counter of F is zero.

Therefore, objects with reference counter greater than zero have to be used as additional roots for the marking phase.

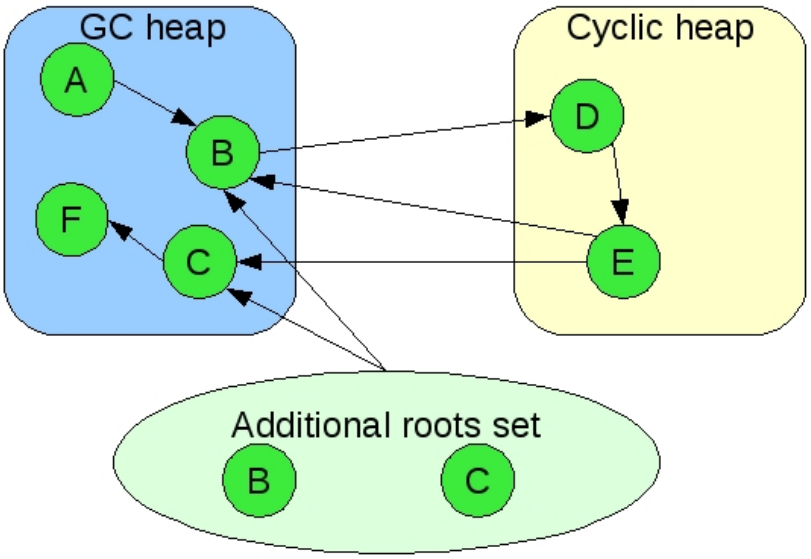


Figure 12: Additional-root set

5.4 Additional-Root Set

As we have decided, objects with reference counter greater than zero will be used as additional roots for the marking phase. Now we have to think about how to manage these roots. The easiest way would be adding an additional search-phase for finding additional roots. However, this would increase pause times, which we want to avoid. Thus, we maintain and manage a set of additional roots.

Figure 12 shows the additional-root set for a small example. Objects B and C are reachable from an object in the cyclic heap, namely Object E. Therefore, these objects have been added to the additional-root set.

An object is added to this set when its reference counter is increased from zero to one. As every reference counter starts from zero, all possible additional roots are added to the set at least once. It is possible that an object is added to the set more than once. This actually does not matter, because during the marking phase objects may be visited more than once too. It is more expensive to check for duplicates.

The problem of duplicates only occurs because we do not remove objects from the additional-root set immediately when their reference counters becomes zero again. This would be too expensive, and it would add additional pause times depending on the size of the set. Before an additional root is considered during the mark phase, the reference counter is checked. If it is zero, the object is removed from the set. Otherwise, all references of this object are processed.

Therefore, every valid additional root is added to the set, and every invalid root is not considered in more than one garbage collection run.

5.5 Delayed Deallocation

An important part of reference counting is to process references of objects which are deallocated. However, in cyclic memory allocation, deallocation is not done explicitly. The memory is just reused. We know when objects are overwritten. It happens at each allocation after the corresponding cyclic buffer has been full for the first time. The newly allocated object has the same internal structure as the previous object, because all objects which are allocated at the same allocation site are of the same type. Therefore, we are able to find the references of the old object. We use the structure of the new object to find the data. Thus, the following steps are required to allocate an object:

1. Find the position in the buffer where the object will be allocated.
2. Use the internal structure of the new object to delete all references of object which is stored at this position.
3. Allocate the new object.

Therefore, before we use the memory again, all references of the old object are deleted and the corresponding reference counters are updated. We call this technique delayed deallocation because an object is deallocated at the moment when its memory is used again.

A consequence of this technique is that some objects might never be deallocated. This zombie objects can keep some objects in the GC heap alive which would be deallocated otherwise. This is a memory leak. However, it cannot grow. As the objects in the memory leak are dead, they cannot be changed by the program, but to let the memory leak grow references from dead objects to new objects have to be created.

Delayed deallocation is also used in [13] to check the correctness of explicit memory management. The best expiration event for an object in explicit memory management would be when the object is is only reachable through one reference. However, the only correct time to deallocate an object is when the last reference to the object is deleted. Programmers who use explicit memory management, first deallocate an object and then delete the reference. The tool presented in [13] waits for some time before it deallocates an object, because then it is possible to check if the object really is dead.

5.6 User input

As we use a system with two coexisting memory management systems, someone has to decide which allocation scheme should be applied on which objects. It might be possible to do this with program analysis, however, for now we let the programmer decide.

By default, every object is allocated into the GC heap. The programmer can decide for each allocation site, if she wants to use cyclic allocation. She also has to tell the system the size of each buffer. Therefore, it is possible to adjust a program step by step.

This was one of the properties we wanted to achieve: If the programmer is able to give some additional information, the system can be improved. Cyclic allocation has a much better runtime behavior than tracing garbage collection, but it cannot be applied on every system. In AGC, beside the good properties of cyclic allocation, it is not necessarily used in the whole program.

5.7 Properties of the System

The preceding sections described the concepts of AGC. In this section we will look at the properties we have achieved. As we stated in Section 5.1, we are using the fastest possible expiration events. This implies that our system can be very fast. On the other hand, the expiration classes of a tracing garbage collector need the most time to be defined, and we need to apply reference counting, which is slow as well.

However, the other expiration class we use is the fastest possible. It is not even necessary to define it at all. To summarize, we will find our system somewhere between high performance and long pause times.

5.7.1 Performance

The only time overhead of a tracing garbage collector is the duration of a collection. The less objects a collection has to consider, the less time it takes. Therefore, every object which is allocated by cyclic allocation reduces collection time. Most objects are short living. It is easy to use cyclic allocation for short-living objects because the programmer can understand all aspects of their purpose and calculate the required size of the cyclic buffers. Without these objects, the garbage collector will run less frequently.

When all dynamic memory is allocated in cyclic buffers, no collections are started at all and pause times are totally eliminated. The allocation time is predictable then. This was one of the properties we wanted to achieve.

Cyclic allocation itself is very fast, its allocation time and its memory consumption are predictable. That was one reason for using it. However, as we already stated before, we had to implement some adaptions in order to use it in combination with tracing, namely reference counting and delayed deallocation.

We already described in 4.2.2 that reference counting is slow when it is done

all the time. Nevertheless, it can be applied when only few references are considered. Modern generational garbage collectors surpass simple tracing collectors most of the time, and they make use of reference counting.

All references have to be checked if reference counting has to be applied. This implies some time overhead. When cyclic allocation is applied on more allocation sites, this overhead will get less, and it will surpass the tracing garbage collector. However, if reference counting has to be done very often, AGC can get even slower. This happens when lots of references in the cyclic heap point to objects in the GC heap. Such situations should be avoided.

The second adaption of cyclic allocation was delayed deallocation. In pure cyclic allocation, the memory of an object is just reused. Now we have to delete all references of the old object. This is some overhead, even if no reference of the object points into the GC heap. A possible optimization would be to check if the whole buffer contains references to the GC heap. If not, delayed deallocation could be skipped.

5.7.2 Correctness

Only garbage collected systems can provide correctness in a sense that no object is deallocated which is still reachable, at the cost of runtime overhead. Sometimes, garbage collection is too expensive. This is the reason why explicit memory management systems are commonly used. In AGC, the programmer can reduce garbage collection step by step and thereby improve runtime behavior and predictability. Nevertheless, the programmer can use AGC in an incorrect way if she gives wrong buffer size information. However, whenever she cannot calculate the buffer size of an allocation site, garbage collection can be used to ensure correctness. The programmer can use cyclic allocation to improve runtime behavior, but she is not forced to use it.

5.7.3 Memory Consumption

For cyclic memory allocation, the amount of memory needed during program runtime is known. The total amount of memory needed is the sum of all cyclic buffers. The memory consumption of a garbage collected memory management is not so easy to determine. For most Tracing garbage collectors, the size of the GC heap can be configured. When the heap is full, a collection is started. When the heap was not big enough, the program crashes. There is some research work on approximating the maximum amount of live memory in garbage collected systems [2], but this is a very difficult goal to achieve for general programs.

In AGC the programmer moves objects step by step from the GC heap to the cyclic heap. Thereby, she makes the total amount of memory more and more predictable.

Most optimizations in computer systems are not for free. In AGC we already discussed the additional information required by the programmer to use cyclic allocation. Moreover, we have some additional memory overhead:

- Reference counter
  - We need a reference counter for every object. This means that every object is at least a bit bigger than it was originally.
- Additional-root set
  - We store our additional roots in a set, which requires some memory too. We fixed the size of the additional set to avoid possible errors in our implementation. In later versions, we may implement the set in a more flexible way.
- Management data for cyclic buffers
  - It is easy to manage a cyclic buffer, but anyway, some data is necessary to do it. This is described in Section 8.5.

AGC can lead to additional memory consumption if the buffer sizes are too big. Our experiments have shown that for most allocation sites, it is easy to find the correct buffer size, and most of the time, it is one.

5.8 Summary

AGC allows a trade-off between correctness and predictability. The programmer controls the system behavior by the amount of information she gives to the system.

6 Other Hybrid Memory Management Systems

Hybrid memory management systems try to combine the advantages of pure systems. Most of the time, garbage collection is one part of such a combination because of its correctness. The second method tries to improve some of the disadvantages of garbage collection. This was the case for AGC too. We wanted to gain predictability and correctness, thus, our second part was a predictable system.

In this section we will discuss three hybrid memory management systems: two combinations of region-based systems with a garbage collector and one combination of a garbage collector with explicit memory management.

6.1 Cyclone - Memory Management

Cyclone is not just a memory management system, but a whole programming language. It wants to provide low-level memory access for the programmer, by still be able to check the correctness of the program at compile-time. The memory handling is described in [15].

The memory management system of Cyclone is a combination of garbage collection and region-based memory management. Memory is typically allocated on the garbage-collected heap. If the programmer wants to have low-level access to some parts of the memory, she can create a region for it. All memory in a region will be deallocated simultaneously.

As we described in Section 5.2, the problem of hybrid systems is how to handle references between differently-managed parts of the heap. In Cyclone, a region is seen as one big object, which is reachable from the stack as long as it is not deallocated. To deallocate a region, the reference from the stack to the region is deleted. The region actually is not deallocated until a collection run. As a region is part of the garbage-collected heap, all references within a live region are tracked in the marking phase of the garbage collector.

The programmer decides when a region can be deallocated. Therefore, dangling pointers and memory leaks might exist. Memory leaks are no problem in Cyclone. All unreachable objects of a memory leak are deallocate by the garbage collector. To avoid dangling pointers, all references which point into a region have to be deleted before the region is deallocated. This is checked at compile-time.

6.2 Digital Mars D - Memory Management

As Cyclone, Digital Mars D is a programming language too. By default, they use garbage collection. Nevertheless, explicit memory management is allowed, as described in [1]. This is necessary, because it wants to use external program code, which is written in a different programming language using explicit memory management.

Explicitly-handled objects, which contain references into the garbage-collected heap, have to be declared. Otherwise, they are ignored at garbage collection as additional roots. Before an object is deallocated again, the declaration has to be removed.

In conclusion, not only deallocation has to be done explicitly, but also the handling of the additional-root set. This can cause memory leaks in the garbage collected heap as it is possible in AGC (described in section 5.5). In AGC, the number of such memory leaks is limited by the buffer size of the according allocation site. However, in Digital Mars D, the number of such memory leaks is unbounded.

Moreover, if the programmer forgets to register an additional root, a dangling pointer might be created, because all objects which are only reachable from this root will be deallocated because they are not found by the marking phase. In AGC, no dangling pointers can point into the GC heap. Therefore, correctness within the GC heap is guaranteed.

6.3 Regions supported by a Garbage Collector

A third combination is presented in [16]. This system uses a region-based memory management system. If the the heap gets full because of growing regions, the whole memory is garbage collected.

To remove internal fragmentation, during garbage collection all live objects are copied into new regions. Objects which were in the same region before garbage collection will be in the same region after the collection run again.

The problem of invalid references mentioned in Section 5.2 exists in this system too. It is solved in the following way: either the deallocation of a region is delayed until it is not reachable anymore, or references which might get invalid are forbidden.

In summary, this memory management system provides a solution for the region-explosion syndrome. If the system runs out of memory, it looks for dead objects in the regions. Therefore, even regions with long-living objects will be partly deallocated sometimes.

Part III

Implementation

7 Implementation Environment

In this section we describe the implementation environment we extended with AGC. First, we will give a short introduction to Java, one of the most popular programming languages. Thereafter, we will present the concept of virtual machines. We have decided to use a particular virtual machine called Jikes RVM. The last part of the section explains this decision and its consequences.

7.1 Introduction of Java

Java is an object-oriented programming language. This means, among other things, that the program does not work on raw memory, but on objects.

Java does not support explicit memory management, it does not provide language constructs for object deallocation.

To run a program on a computer, it has to be translated from human-readable source code, into machine-readable native code. This procedure is called compilation. The translator is called compiler. Different computers and operating systems (called platforms) use different native code. If it should be possible to run a program on more than one platform, it has to be compiled for each of the platforms.

Java goes another way. Java is not compiled into native code, but into some kind of intermediate code, called byte code. It is not possible to run byte code directly on a machine. Every platform must have a program called virtual machine (VM), which can execute Java byte code. This is illustrated

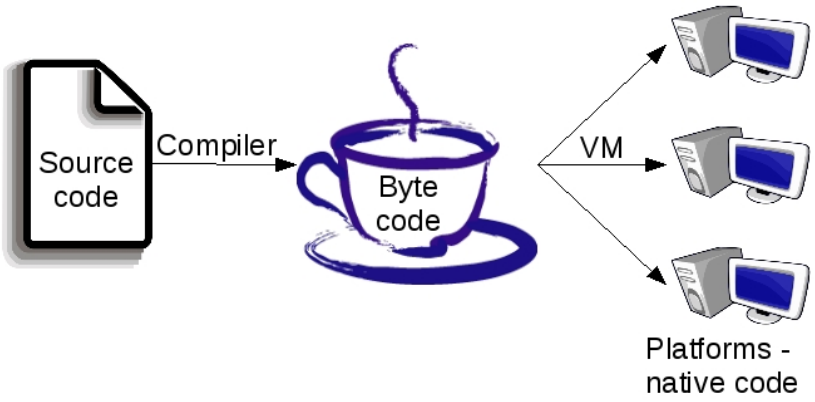


Figure 13: Java: From source code to execution

in Figure 13. Source code is compiled into byte code by a compiler. The byte code can be executed on different platforms by a VM. More information about the programming language Java can be found in [14].

7.2 Virtual Machine

Java programs typically run on a virtual machine. In this section, we will deal with the concept of virtual machines and we will specify all important components of it. The main task of a Java virtual machine is to execute byte code. The Java byte code is a sequence of commands which have to be executed one after the other. These commands are very similar to native-code commands. Together with the byte code, the virtual machine gets some additional information about the program. This is called “meta data”. Byte code and meta data are packed together in a Java class-file.

There are two ways to execute Java programs in a virtual machine. In modern VMs, both techniques are used. The first technique is called interpreting. As byte code and native code are very similar, it is possible to read one command of the sequence and to execute it immediately. This is done by an interpreter. To start execution, nearly no preliminary work has to be done. Of course, this is not as fast as native execution, but the main purpose of Java programs

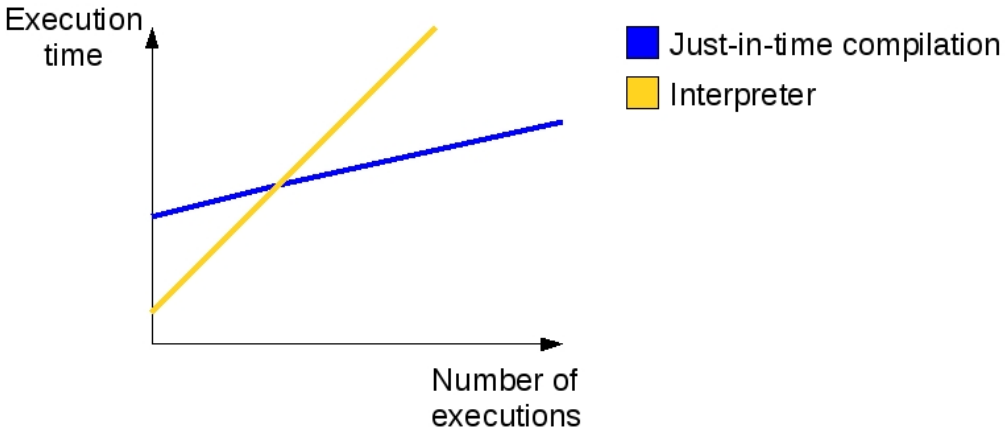


Figure 14: Just-in-time compilation vs. interpreting

is correctness rather than high performance.

The second technique is called just-in-time compilation. Before a part of the byte code, called method, is executed, it is compiled into native code, which can be used for further executions. When a method is started only once, an interpreter is faster. However, when it is started more often during program execution, just-in-time compilation has higher performance. This is visualized in Figure 14. For the first executions, the interpreter is faster because it needs no preliminary work. When the same method is executed more often, just-in-time compilation surpasses interpreting.

Furthermore, every Java virtual machine delivers some kind of implicit memory management, most of the time a garbage collector. We will change the memory management of a virtual machine to use AGC. For further information about the Java virtual machine we refer to [17].

7.3 Jikes RVM

First of all, we had to decide which Java virtual machine we wanted to extend with AGC. We decided to use the Jikes research virtual machine developed by IBM, which is described in [3]. Jikes has some great benefits which led to

this decision:

- Jikes is called a research virtual machine because it supports efficient research about all parts of a virtual machine. For memory management, it even contains a small scripting language to test the memory management independently of the rest of the virtual machine. It also provides all required constructs for AGC, like reference counting, an efficient tracing implementation, and ways to work on a split heap.

- Unlike other virtual machines, which are written in C or C++, Jikes is written in Java. This is described in [4]. It is very convenient to develop applications in Java and there exist powerful tools to do this. Furthermore, the design of Java implies well structured programs, which are easier to understand than programs written in C. This is an important feature, because to extend a program, one first has to understand it. Virtual machines are complex applications and it is always hard to dig into unfamiliar programs. Although Jikes is a Java program, it does not run in a virtual machine itself. To make this possible, some tricks are used. However, this is not important for our system.

Jikes also has some drawbacks, which we fortunately are able to deal with:

- Jikes only uses just-in-time compilation, because its first intention was to execute server programs. Most server programs run for a very long time and just-in-time compilation pays off in the majority of methods. The problem is that a just-in-time compiler is more difficult to change than an interpreter. We had to change it because we wanted to support a second kind of allocation method, cyclic allocation. For this purpose we utilize some special mechanisms, which are used by Jikes to be executed without a virtual machine.

- Jikes uses the same memory management for itself and for the Java program it executes. When we use AGC for the user program, Jikes uses AGC for itself. It is not easy to use cyclic allocation for Jikes. Thus, Jikes only uses Tracing garbage collection for its own memory.

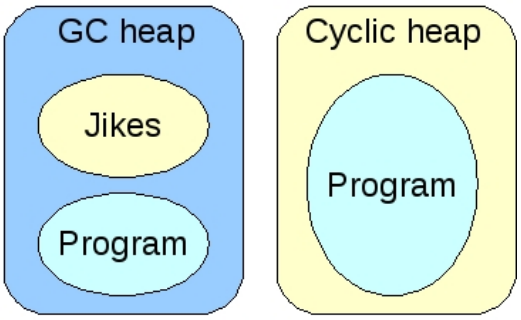


Figure 15: Memory partition in Jikes

The memory partition is illustrated in Figure 15. The GC heap contains both Jikes objects and program objects, the cyclic heap only includes program data.

We are able to solve the first problem in a very nice way, but due to the second problem, it is impossible to only use cyclic allocation at the moment.

8 AGC: Implementation

In this section we will deal with our implementation of AGC as memory management of Jikes. First we will describe the initial system, thereafter, we will explain all changes and extensions.

8.1 Initial System

Jikes provides many implementations of garbage collection, like Tracing, reference counting and generational garbage collection. Jikes is configured to use one of them. One part of AGC is a Tracing garbage collection memory management system. Thus, we start with the Tracing configuration.

Jikes uses some extra partitions of the heap, which are not handled with garbage collection, for example, one segment of the heap contains the byte code of the program and of Jikes itself, another segment is for permanent memory. It is necessary for Jikes that some parts of its memory are not handled by the garbage collector, for example the memory which is required by the garbage collector itself.

8.2 User Input

The developer of the program has to provide the information about how to allocate objects. For now, she can do this with annotations. Annotations are a feature of Java to add meta information to methods. We called the new annotation "CyclicAllocation". Whenever the just-in-time compiler wants to compile a method, it checks if it is annotated with this annotation. If so, all allocation commands in this method are translated into cyclic allocation commands.

It is not possible to annotate a single allocation command itself, which would be even more convenient. If there is more than one allocation site in one

Algorithm 1 An annotated method

```
1 @CyclicAllocation (bufferSize = 2)
2 private Object createObject ()
3 {
4     return new Object ();
5 }
```

method, we are not able to distinguish them. Therefore, we only allow one allocation site in an annotated method.

The annotation contains one parameter, called "bufferSize". This parameter specifies the size of the cyclic buffer required for this allocation site.

Algorithm 1 is an example of an annotated method. "createObject" is the name of the method, "new Object()" is the allocation command, "@CyclicAllocation(bufferSize=2)" is the annotation. A buffer of size two is set for this allocation site.

In order to use cyclic allocation, the programmer has to extract every allocation site she wants to change into a small method, which she annotates afterwards.

8.3 Memory Partitioning

We need an additional segment in our heap which is managed by cyclic allocation. It is important to check if an object lies in the cyclic heap or in the GC heap. Therefore, we use a contiguous piece of the heap for the cyclic heap as well as for the GC heap. It is very fast to check if the position of an object is between the start and the end of a heap segment.

At the moment, both heap segments have the same fixed size for all programs. It is possible to calculate the size of the cyclic heap of a specific program. Anyway, this would require some kind of program analysis, which is not available at the moment.

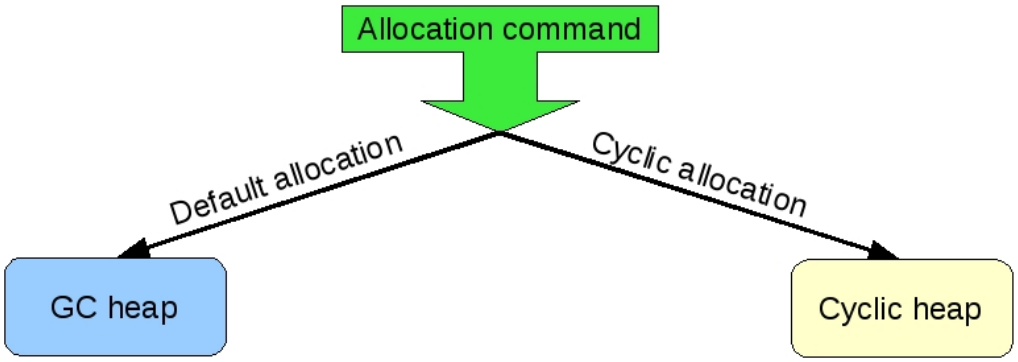


Figure 16: Jikes allocation command

8.4 Allocation

The allocation command in Jikes already had a parameter to specify the allocation method. For example, some possible values were “defaultAllocation”, “immortalAllocation” or “codeAllocation”. We added the allocation method “cyclicAllocation”. When the just-in-time compiler detects a "CyclicAllocation" annotation, it changes the allocation method of the specific allocation site from “defaultAllocation” to “cyclicAllocation”. Thereby we can allocate each object into the correct heap segment.

The treatment of an allocation command is shown in Figure 16. Objects with the “defaultAllocation” parameter are allocated in the GC heap, object with the “cyclicAllocation” parameter are stored in the cyclic heap.

8.5 Cyclic Allocation

The implementation of cyclic allocation can be split up in three important parts:

1. How to find the cyclic buffer of an allocation site?
2. How to find the next slot in the cyclic buffer?
3. Is delayed deallocation required before object allocation?

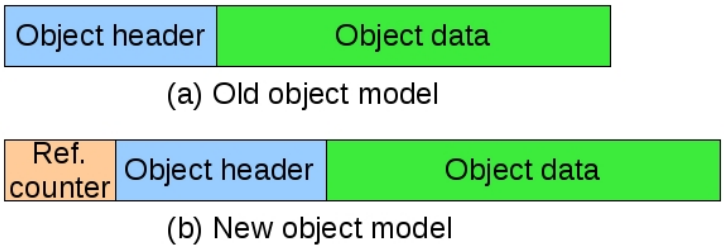


Figure 17: Changes in the object model of Jikes

Every allocation site, which uses cyclic allocation, gets a unique ID by the just-in-time compiler. The implementation of the cyclic allocation contains a look-up table, which contains the buffer size, the position of the buffer in memory, the position of the next slot to use in the buffer and a flag to indicate if delayed deallocation is already necessary for this buffer.

This look-up table contains the answer to all three questions. Each allocation site can find its entry in the table by its ID. The entry contains all information to deal with all three questions. After an allocation, the entry has to be updated. The position of the next slot changes at every allocation, the delayed-deallocation flag changes just once.

8.6 Object Model

An object does not only contain program data, but also some information for the virtual machine itself. This information is organized in the so-called object header. For example, it contains the mark bit required for a Tracing garbage collector. An object model describes which information the object header contains and how the information is organized.

AGC requires a reference counter in all objects of the GC heap. Nevertheless, we extended all objects with a reference counting field, because Jikes does not support the use of multiple models.

Figure 17 shows the changes in the object model. The reference counting field was added at the begin of every object. Thus, it is easy to find independently

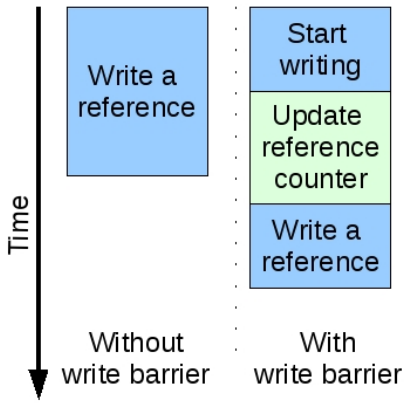


Figure 18: The write barrier for reference counting

of the object size.

8.7 Write Barrier

A write barrier is a generalization of reference counting. The barrier allows to execute additional code before a new reference is written. The write barrier for reference counting is illustrated in Figure 18: Without the write barrier, only the value of a reference in an object is changed. When a write barrier is applied, before the actual writing is done, the reference counter is updated.

Not all objects require references counting. Therefore, we check if the source object lies in the cyclic heap, and if the target belongs to the GC heap. Only if both conditions are true, the update of the reference counters is started.

A write barrier implies some overhead, even if nothing is done by it. This will be shown later in some baseline benchmarks. Anyway, it enables us to replace garbage collection by cyclic allocation step by step, which permits spatial predictability, performance enhancement, while still allowing garbage collection on critical parts of the program. Research results about the overhead of write barriers can be found in [9].

8.8 Additional-Root Set

We use a list to manage the additional-root set. An object is added to the list if its reference counter is incremented from zero to one in the write barrier described in Section 8.7. It is possible that the additional-root set contains an object more than once. Anyway, this does not change the result of the marking phase during garbage collection.

An object is not removed from the list immediately when its reference counter gets zero again. It would take too much time to find an entry in the list. If the reference counter of an object is zero, it is removed when the list is processed during collection. It is not enough for an object to be contained in the additional-root set to avoid deallocation, the reference counter has to be greater than zero too.

8.9 Delayed Deallocation

When the cyclic buffer of an allocation site gets full for the first time, the delayed-deallocation flag is set. For all further allocations into this buffer, delayed deallocation is required. In a delayed deallocation, to deallocate an object, all references are read from the object. Afterward, these references are deleted and the the reference counters of the targets are updated.

8.10 Marking Phase

To ensure that all live objects are marked during the marking phase, we could add our additional roots to the original root set. However, in Jikes it is possible to add custom phases to a garbage collector. Therefore, we added an additional marking phase to process the new set of additional roots. It works in the same way as the original marking phase, but with a different root set. After both marking phases, the sweeping phase is started.

As stated in Section 8.8, an object is only processed as additional root if its reference counter is greater than zero. Otherwise, the object is removed from the additional-root-set.

Part IV

Experiments

9 Experiments and Results

In this section we will present some benchmarks we made with our system, and we will compare it with other systems, namely the Tracing garbage collector and the generational garbage collector, both implemented in Jikes. The heap size of the virtual machine is 20MB. In all test runs of AGC, the size of the cyclic heap is 20% of the size of the total heap.

First we will measure the overhead of write barriers of our system. To do this, we execute some applications of the Dacapo benchmark suite, version 2006-10, without any changes. The Dacapo benchmark suite is described in [8].

Afterwards, we will start some programs which take use of cyclic allocation. The JLayer MP3 encoder and the monte-carlo application of the Grande Java Benchmark Suite. This benchmark suite is described in [18]. We will start annotated and unannotated versions of the programs. For the monte-carlo application, we will show that most of the performance boost of cyclic allocation comes from few allocation sites.

At last we present a small self-written program, consisting of just one cyclic allocation site which is executed one million times. This program is ideal for cyclic allocation, and the results show the capabilities of our method.

The tests were performed on system with a 2.16 GHz Intel Core Duo processor and 4 GB memory.

9.1 Dacapo Benchmark Suite

The Dacapo benchmark suite is a very popular Java benchmark suite. We executed the programs of Dacapo to measure the overhead of AGC in an unchanged system. Thus we started the tests with both AGC and a Tracing garbage collector. As the most important source of overhead is the write barrier, we additionally ran the benchmarks with a generational garbage collector. This garbage collector uses a write barrier too. Nevertheless, it is used in most general-purpose virtual machines.

We started seven programs of Dacapo, each program was executed five times on all three system. Table2a contains the average results of the test runs. The first column contains the names of the benchmark programs, the second, the forth and the sixth column contain the runtime of the tests in milliseconds. We defined the runtime of the Tracing garbage collected system as our baseline. The third, the fifth and the seventh column contain the runtime of the benchmarks in percent according to this baseline. Table 2b shows the standard deviation.

The overhead is between 5% and 56%. An overhead of 56% is a lot. On the other hand, the programs with the most overhead in AGC, namely the programs "bloat" and "pmd", also have an enormous overhead in the system using the generational garbage collector. This means that these programs suffer a lot from the write barrier.

The write barrier can be very expensive. The following benchmark tests will show the results of AGC executing adapted programs.

9.2 JLayer MP3 Encoder

We used the JLayer MP3 encoder[1] to test our system because it consists of one big cyclic calculation. Moreover, MP3 encoding is used in real-time applications, which is one of our targets. As we are only interested in the

[1] www.javazoom.net/javalayer/javalayer.html

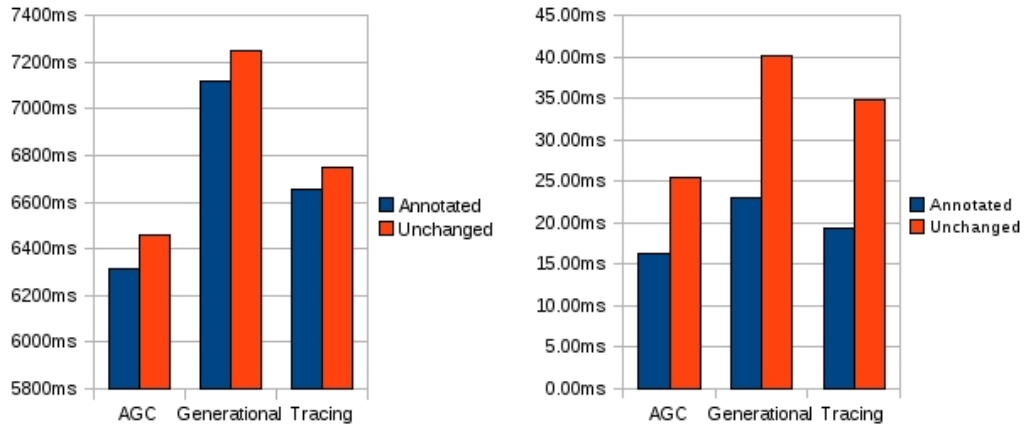
	AGC (in ms)	AGC (in %)	Gen. GC (in ms)	Gen. GC (in %	Tracing (in ms)	Tracing (in %)
bloat	122102ms	156.01%	95257ms	121.71%	78264ms	100.00%
chart	82904ms	112.19%	73211ms	99.07%	73896ms	100.00%
fop	11379ms	105.07%	10639ms	98.24%	10830ms	100.00%
jython	94117ms	125.27%	61095ms	81.32%	75129ms	100.00%
luindex	61017ms	117.89%	54932ms	106.14%	51757ms	100.00%
pmd	84537ms	132.12%	75615ms	118.18%	63985ms	100.00%
xalan	61465ms	119.83%	53635ms	104.56%	51295ms	100.00%

(a) Average results

	AGC	Generational GC	Tracing GC
bloat	63ms	142ms	42ms
chart	51ms	92ms	84ms
fop	18ms	68ms	13ms
jython	65ms	5ms	65ms
luindex	71ms	75ms	94ms
pmd	115ms	104ms	60ms
xalan	137ms	199ms	188ms

(b) Standard deviation

Table 2: Results of the Dacapo benchmark suite



(a) Average results

(b) Standard deviation

Figure 19: Results of the JLayer mp3 encoder test runs

memory behavior of the program, we removed the calculation unit of the encoder. This improves the significance of the results, but it does not favor a particular system.

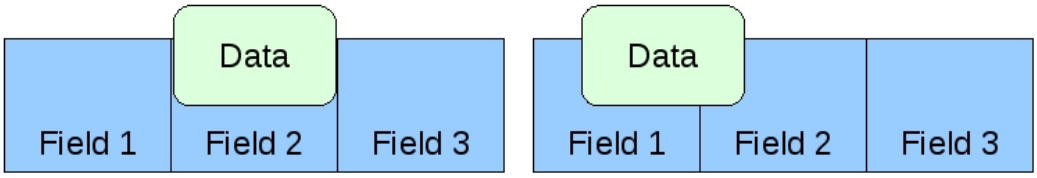
We started an annotated and an unchanged version of the MP3 encoder in all three test systems. Every combination was executed 20 times.

The results can be seen in Figure 19. They are very surprising, because AGC is faster than the Tracing even in the unchanged version. The reason for this is the way how a computer can use its memory, as explained below.

9.2.1 Alignment

Data loaded from the memory into the CPU always has the same size. When the size of the data is greater than this portion, more than one operation is required to load it. Moreover, a portion cannot start anywhere in memory. Therefore, it is possible that data has the size of one portion, but still it cannot be loaded in one operation.

This is illustrated in Figure 20: The three fields represent the portions of



(a) Aligned data (b) Unaligned data

Figure 20: Alignment of data in memory

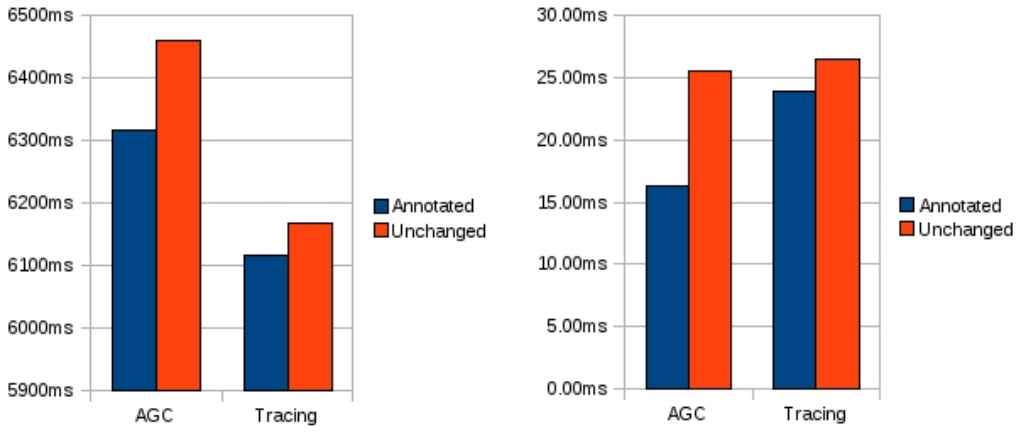
memory which can be loaded from memory in a single operation. "Data" represents the data which should be loaded. It is possible to load all data in a single step. This can be done in the settings of Figure 20a. If the data is not aligned, as one can see in Figure 20b, two operations are required, because Field1 and Field 2 have to be loaded.

This phenomenon happened in the test runs of the mp3 encoder. Lots of data are used which have exactly the size of one portion. Anyway, the alignment of the data in the Tracing system was not good. Lots of load-operation required two steps. Every object in AGC has some additional header bytes for reference counting. These additional bytes fixed the alignment of the data and increased the performance of the system.

9.2.2 Analysis

Now we are able to analyze the results. By annotating the program we slightly improved the performance of all three systems. However, these changes do not favor AGC. The garbage collector is not triggered very often during the execution of the mp3 encoder.

To test the effects of bad alignment, we started the benchmarks of the tracing system again, but this time we added some bytes to the header of every object, as we did it in AGC. The results are presented in Figure 21. AGC benefits much more of the annotations than the changed tracing system.



(a) Average results

(b) Standard deviation

Figure 21: JLayer mp3 encoder: AGC vs. tracing with extended header

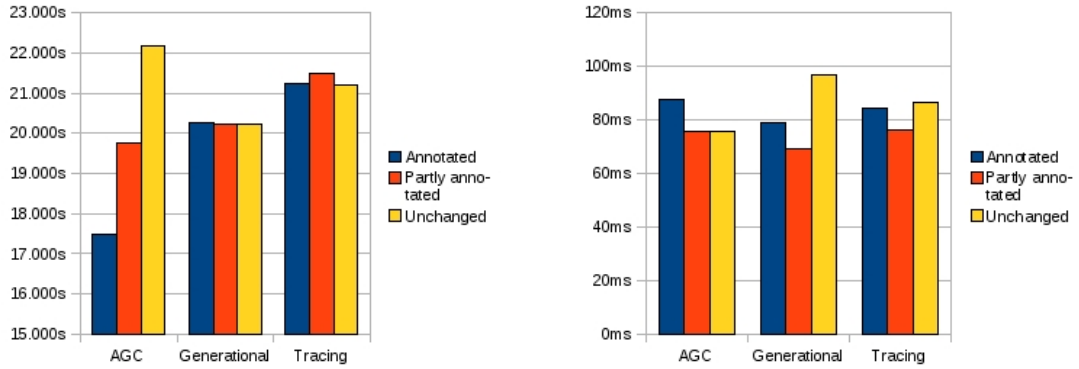
9.3 Monte Carlo

As the MP3 encoder, calculations with a monte-carlo method run regularly in cycles. The original program was taken from the Grande Java Benchmark Suite, described in [18]. It was not difficult to annotate the whole application.

Again, we test the same three memory management systems. This time, we start with three implementations of the monte-carlo program: An unchanged version, a fully-annotated version, and one implementation, which contains annotations for one allocation site and its sub-allocation sites. The sub-allocation sites are called by the main allocation site.

Every system/program combination was executed 20 times, Figure 22 shows the results. The first three columns are the result of AGC, the second three columns present the results of the generational garbage collector, and the third three columns illustrate the results of the tracing garbage collector. The first column stands for the results of the fully annotated program, the second column represents the party annotated program, and the third column stands for the original program.

As one can see, the annotation does not change a lot for the purely garbage-



(a) Average results

(b) Standard deviation

Figure 22: Results of the monte-carlo benchmark

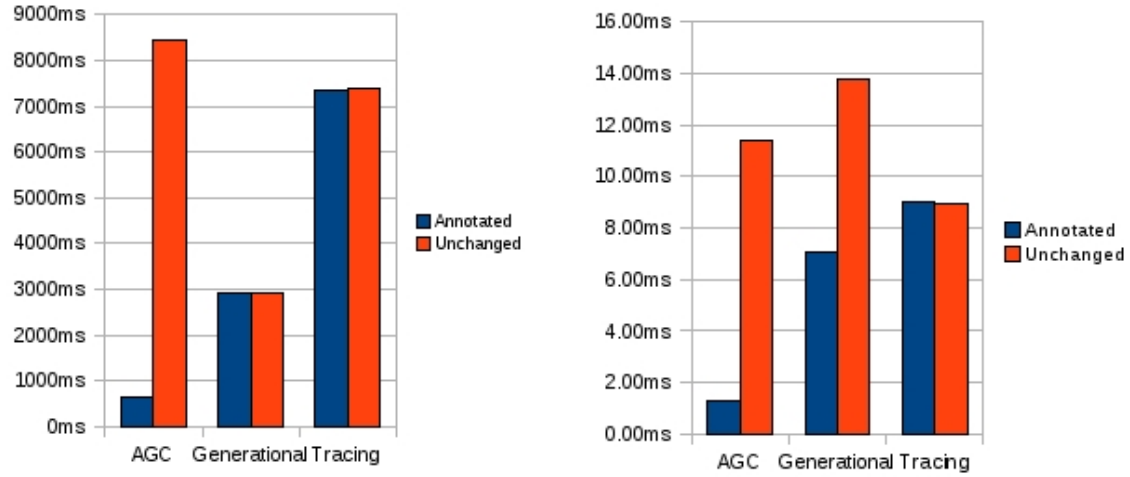
collected systems, but with AGC, the annotated program is 22% faster. However, half of the performance increase is gained in just one allocation site. This shows that even few annotations can have a great gain.

The generational garbage collector is faster than tracing because the program has some permanent data, but all other objects have a very short live-time. The generational garbage collector processes the permanent data only once, whereas tracing checks it at every collection run.

We also counted the number of garbage collections required for the annotated version of the monte-carlo benchmark. The tracing system requires 13 collections, whereas AGC only starts the garbage collector twice. This reduction is the main source of the runtime improvement of AGC.

9.4 Cyclic Allocation Program

This benchmark is an extreme corner case. Anyway, it shows the improvement a program can have by using AGC. It just allocates an object one-million times, without using the object. In AGC, the object is allocated in a cyclic buffer, the other systems have to collect the dead objects in a collection run. Again, we started all three systems with both an annotated and



(a) Average results

(b) Standard deviation of the results

Figure 23: Results of the cyclic allocation program

an unchanged version.

The results are illustrated in Figure 23. The first pair of columns stands for AGC, the second for the generational garbage collector, and the third for the Tracing garbage collector. The first column of every pair represents the annotated program, the second column stands for the unchanged program.

The annotation improves AGC enormously, because no garbage collection is required at all. Therefore, it is ten times faster than the Tracing system. Moreover, the AGC results are very constant, as the standard deviation is just 1 millisecond.

The difference between the two systems can be increased by filling the heap with permanent data. Garbage collection is then started more often. This was done in a more extreme test case. AGC was 40 times faster than Tracing, by just 100.000 allocations, whereas the permanent data did not affect AGC at all. The runtime of the system with generational garbage collection was not affected too, because it only collects the nursery and ignores the permanent data in the mature space.

9.5 Summary

The benchmarks showed that AGC has some overhead because of the write barrier. Nevertheless, when annotations are used, the performance of AGC improves. The system benefits a lot if short-living objects are handled by cyclic allocation, because the number of garbage collections is reduced significantly. The system even benefits if permanent data is handled by cyclic allocation: Permanent data is the big shortcoming of Tracing. It has to be processed in every collection run, but it yields nothing. If permanent data is ignored by the garbage collector, more memory is deallocated in less time.

Part V

Conclusions

10 Conclusions

This section concludes the master thesis. The first part of the thesis introduced some important aspects of memory management. Afterward, we described the expiration class analysis, a powerful way to analyze existing and to develop new memory management systems. It is easy to use, and the results are very revealing.

We discovered that explicit memory management has the problem of correctly defining its expiration events. Garbage collection has either a lack of expiration class management or expiration events are hard to detect. Region-based memory management and cyclic allocation have the potential to outperform explicit memory management, but they cannot be used for general purpose.

In the following sections, we introduced the concepts of AGC, our new memory management system, which we developed on the knowledge we gained by the expiration class analysis. We combined the correctness of a Tracing garbage collector with the predictability and performance of cyclic allocation. First we gave a conceptual overview, then we described the virtual machine, which we extended with AGC, and at last we presented our implementation.

In Section 9, we presented the benchmarks run with AGC. The results show that AGC starts with some overhead, but when the programmer gives additional information on the structure of the program AGC beats garbage collected systems in performance and predictability. The number of garbage collections can be reduced significantly.

10.1 Future work

Our system AGC can be improved in several ways:

1. Sometimes it is not easy to annotate a program, because the buffer size does not only depend on the allocation site, but also on the number of parent allocation sites. An allocation site can only be called if its parent allocation site was called before. It is difficult to do this by hand, but the buffer size of most allocation sites could be calculated at compile time automatically. This would make the use of AGC more convenient for the programmer.

2. At the moment, only single-threaded applications are supported by AGC. A future step would be to support multi-threaded applications. There are no obvious obstacles to do this, but at the moment, we were concerned more about the theoretical aspects, which are independent of the number of threads.

3. A third step would be to try other garbage collectors instead of tracing. For instance, a generational garbage collector already uses a write barrier. Therefore, the reference counting of AGC would introduce less overhead than in the current configuration. Moreover, permanent memory is not considered in all collection runs, which can reduce pause times additionally.

4. Furthermore, some allocation sites cannot be handled by cyclic allocation. Other expiration events and expiration classes could be used for this kind of objects. It is possible that region-based systems are able to deal with the shortcomings of cyclic allocation.

References

[1] Memory management - explicit class instance allocation, July 2009. `http://www.digitalmars.com/d/2.0/memory.html`. 45

[2] E. Albert, S. Genaim, and M. Gómez-Zamalloa Gil. Live heap space analysis for languages with garbage collection. In *ISMM '09: Proceedings of the 2009 international symposium on Memory management*, pages 129–138, New York, NY, USA, 2009. ACM. 42

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238, 2000. 49

[4] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing jalapeño in java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 314–324, New York, NY, USA, 1999. ACM. 50

[5] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2):171–183, 1989. 25

[6] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. *SIGPLAN Not.*, 38(1):285–298, 2003. 27

[7] D. F. Bacon, P. Cheng, and V. T. Rajan. A unified theory of garbage collection. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 50–68, New York, NY, USA, 2004. ACM. 25

[8] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM. 59

[9] S. M. Blackburn and A. L. Hosking. Barriers: friend or foe? In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 143–151, New York, NY, USA, 2004. ACM. 56

[10] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: fast garbage collection without a long wait. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 344–358, New York, NY, USA, 2003. ACM. 26

[11] G. E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, 1960. 22

[12] S. S. Craciunas, C. M. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger. A compacting real-time memory management system. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 349–362, Berkeley, CA, USA, 2008. USENIX Association. 15

[13] D. Gay, R. Ennals, and E. Brewer. Safe manual memory management. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 2–14, New York, NY, USA, 2007. ACM. 39

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. 48

[15] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, New York, NY, USA, 2002. ACM. 44

[16] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 141–152, New York, NY, USA, 2002. ACM. 46

[17] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, 1999. 49

[18] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of java grande benchmarks. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 72–80, New York, NY, USA, 1999. ACM. 59, 64

[19] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960. 20

[20] H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 15–30, New York, NY, USA, 2007. ACM. 28, 29, 30

[21] G. Salagnac, C. Rippert, and S. Yovine. Semi-automatic region-based memory management for real-time java embedded systems. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 73–80, Washington, DC, USA, 2007. IEEE Computer Society. 28

[22] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Not.*, 19(5):157–167, 1984. 25