Linköping Studies in Science and Technology
Dissertation No. 828

Memory Efficient Hard Real-Time Garbage Collection

Tobias Ritzau

Department of Computer and Information Science
Linköping University, SE-581 83 Linköping, Sweden

Linköping 2003

Abstract

As the development of hardware progresses, computers are expected to solve increasingly complex problems. However, solving more complex problems requires more complex software. To be able to develop these software systems, new programming languages with new features and higher abstraction levels are introduced. These features are designed to ease development, but sometimes they also make the runtime behavior unpredictable. Such features can not be used in real-time systems.

A feature that traditionally has been unpredictable is garbage collection. Moreover, even though a garbage collector frees unused memory, almost all such methods require large amounts of additional memory. Garbage collection relieves developers of the responsibility to reclaim memory that is no longer used by the application. This is very tedious and error prone if done manually. Since garbage collection increases productivity and decreases programming errors, developers find it attractive, also in the real-time domain.

This thesis presents a predictable garbage collection method, *real-time reference counting*, that increases memory efficiency by about 50 % compared to the most memory efficient previously presented predictable garbage collector.

To increase performance, an optimization technique called *object ownership* that eliminates redundant reference count updates is presented. Object ownership is designed for reference counters, but can also be used to increase the performance of other incremental garbage collectors.

Finally, a *static garbage collector* is presented. The static garbage collector can allocate objects statically or on the runtime stack, and insert explicit instructions to reclaim memory allocated on the heap. It makes it possible to eliminate the need for runtime garbage collection for a large class of Java applications. The static garbage collection method can also be used to remove costly synchronization instructions. Competing static garbage collection methods with reasonable analysis time are restricted to stack allocation, and thus handle a smaller class of applications.

To the one
who invented
icecream

— I don't know half of you half as well as I should like;
and I like less than half of you half as well as you deserve.

Bilbo Baggins

Acknowledgments

Jag vill börja med att tacka min handledare Peter Fritzson för handledning, stöd och för det förtroende du har visat mig. Jag vill också tacka min bihandlerade Roger Henriksson för alla givande diskussioner vi har haft och Boris Magnusson för inspiration och en arbetsplats.

Ett varmt tack går också till hela PELAB och till programvarugruppen vid LTH för den otroligt trevliga och inspirerande forskningsmiljön jag har fått vara en del av. Ett speciellt tack vill jag skänka till Bodil Mattson-Kihlström för att du har hållit reda på mig.

Jag vill också tacka Gösta Sundberg, Mathias Hedenborg och Ulf Cederling vid Växjö universitet. Det var ni som fick mig att fundera på forskarstudier och de var ni som gjorde det möjligt för mig att påbörja dem. Ett extra varmt tack vill jag ge min vän och arbetskamrat Jesper Andersson. Hoppas att jag har hjälpt dig lika mycket som du har hjälpt mig.

Dessutom vill jag tacka Peter Fritzson, Christoph Kessler och Roger Henriksson för de kommentarer och förbättringar ni har bidragit med efter att ha läst avhandlingen.

Jag vill också passa på att tacka alla de lärare som under åren har fått mig att vilja lära mig mer. Speciellt vill jag tacka Leif Svensson och Bengt-Göran Magnusson vid Dalslundsskolan; Tomas Träpja, Klas Nilsson och Bert Konsberg vid Pauliskolan; samt Ulf Söderberg vid Växjö universitet.

Sist och främst vill jag tacka mamma, pappa, Anja, Annette, Atlas, Medes, Musen, Bamse och ljusstrålarna Elsa och Axel för att ni finns, för att ni står ut med mig och för att ni gjort mig till den jag är. Och ett sista varmt tack skänker jag till alla mina vänner.

Frid och lycka!

Tobias Ritzau
Lund, den 22 april 2003

This work has been support by the ECSEL research school, the EC funded JOSES and HIDOORS projects, the ESA funded AERO project, and Växjö University.

Contents

List of Figures

— It's a dangerous business
going out your front door.

Bilbo Baggins

Chapter 1

Introduction

This thesis presents work in the area of automatic memory management for hard real-time and embedded systems. The motivation of the thesis is to be able to develop hard real-time and embedded systems using modern languages. Since these languages commonly use automatic memory management or garbage collection (GC), which traditionally has had an unpredictable runtime behavior, we could either try to eliminate the need for GC using manual techniques, or we could develop GC techniques for these systems. Since GC is such a powerful tool to eliminate memory related programming errors, we decided to develop techniques to use GC in hard real-time and embedded systems. During this work three other GC techniques for these systems have been published. The main advantage of our work compared to the other three is that memory utilization efficiency increased by about 50 %. We have also developed an optimization for incremental garbage collectors and a static garbage collector that aims to eliminate the need for runtime garbage collection.

1.1 Perspective

Once upon a time, programming required a deep knowledge of how the machines were constructed and the programmers had full control of the execution of the system. Charles Babbage became the first programmer when he programmed his difference machine in 1822. It was programmed by exchanging the gears that performed the calculations. More than 100 years later in about 1945 Konrad Zuse developed Plankalkül [BW72], the first programming language. Unfortunately, most work was lost or confiscated in the aftermath of World War II and the work was not published until 1972. Plankalkül was used to program the Z3, the first universal computer in the world [Roj98].

Contemporary computers were more like calculators, and the calculations where input by punching holes in paper tapes (Z3 and Colossus) or even by making physical changes to the hardware (ENIAC). In 1945 John von Neumann published the EDVAC report [vN45] and Alan Turing published the ACE Report [TCD86]. Both came to the conclusion that programs should be stored in memory in the same way as data was. This was the birth of the computer architecture that is still used today. In 1949, Short Code [Sch88] was introduced by John W. Mauchly, it was the first programming languages for the new generation computers.

Programming languages has since evolved, adding features like recursion, pointers, dynamic memory management, garbage collection, structured programming, object-orientation, etc. Many of these features have become natural parts of programming languages, and most developers can not write a non-trivial program without them. These features makes programming less error-prone, and more complex systems can be implemented. However, with a higher level of abstraction, the control of the applications runtime behavior is lost. When developing real-time systems, i.e. systems whose correctness is not only dependent of their output but also on their timing, it is crucial that the runtime behavior can be predicted.

A conflict occurs when real-time systems become increasingly more complex. Modern languages would certainly ease development and produce more stable systems, but the control of the runtime behavior is lost. The features of modern languages are not the problem, it is the way they are implemented that cause problems. Their implementations usually try to optimize average performance, and not worst case performance as is required in real-time systems. This thesis focuses on automatic memory management of real-time and embedded systems, and presents techniques to make it predictable and still efficient.

1.2 Problem Definition

To be able to maintain full control of the runtime behavior of a system, it must be possible to predict the amount of resources (e.g. CPU time and memory) that is required for any (virtual) machine level instruction and for all runtime system work. Note that using such a system does not prevent writing an unpredictable application. An example is an application that waits for external events, e.g. input from a user. First, it is not always possible to know when the event occurs, and second the data passed with the event may be unknown. Thus, developers must still follow rules to handle such cases.

Early implementations of new languages are typically designed to be easy to implement and prove correct. Then follows optimizations for the average case, which is commonly interactive window based applications

or possibly servers. Techniques that are optimized for such systems are seldom appropriate for hard real-time and embedded systems, because their target systems need not be predictable and they have much more memory resources available.

To be more specific, garbage collection algorithms may be designed to interrupt the application for short time periods in the general case, but it need not be guaranteed that it will collect all garbage memory before the system runs out of memory. If the memory runs out, the system can be stopped to collect the remaining garbage memory. Such stop may take a second or two, but that does not matter to these systems. Unfortunately many such techniques are called real-time garbage collectors, which is confusing. Another problem with garbage collectors is that they consume very much memory. The runtime systems that use real-time garbage collectors that guarantee memory availability need about 70 % of the system memory for internal use, which leaves about 30 % for the application. A large contribution to the overhead comes from the memory that is needed to allocate objects while the garbage collector collects garbage memory. This alone typically causes an overhead of about 50 %.

The garbage collector is not the only part of a runtime system that needs to be redesigned to make it predictable. Examples of other parts that need attention are thread support, synchronization, messaging, and some complex instructions. This is, however, out of scope for this thesis.

1.3 Contributions

The main contributions of this work can be divided into three parts.

Hard Real-Time Garbage Collection Real-Time Reference Counting, or RTRC for short, is a real-time garbage collection technique based on reference counting. Its main advantages are that the memory usage efficiency is increased with approximately 50 % compared to competing techniques, and that the synchronization that is required only locks the system for a few machine instructions.

Garbage Collection Optimization Object ownership is an optimization technique for incremental garbage collectors. It optimizes the code needed to maintain the garbage collection state when one reference is known to keep an object alive. That is, the garbage collector can ignore other references to an object if one reference is known to refer to the object during the life times of the other references. It can typically be applied to the formal parameters of methods, since the passed objects are commonly referred to by the calling method. It can also remove overhead caused by temporary variables. The optimization can work in two modes, either a conservative

mode, or a more aggressive mode that also requires some memory overhead in each object. In the conservative approach the overhead caused by the write-barrier is completely eliminated, and using the other approach the execution time performance of the write-barrier increases with up to 75 %.

Static Garbage Collection A problem that remains is how to predict the memory usage of an application. No garbage collector can keep a system from running out of memory if it uses more memory than is available. By using static analysis, the task of predicting memory usage can be simplified. A static garbage collector determines when objects can be reclaimed at compile-time. This information can be used to optimize allocations and to insert explicit free instructions. As a side effect, the task of calculating how much memory an application requires is greatly simplified. The static garbage collector presented here is not limited to stack allocation, as other static garbage collectors with a reasonable analysis time. By using the static garbage collector presented in this thesis, many real-time systems can be designed to execute without a runtime garbage collector, which increases performance and the ease of proving them correct.

1.4 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2 gives a short introduction to real-time systems. Common approaches are presented, and problems are discussed.

Chapter 3 presents basic garbage collection techniques that most new garbage collectors are based upon. The chapter is concluded with a discussion of which technique to use as the basis of our real-time garbage collector.

Chapter 4 presents the *Real-Time Reference Counting* technique. First the problems of the standard reference counting technique are presented, followed by their solutions. The full design is then presented with a discussion of how it can be implemented. The chapter is concluded by a theoretic comparison in both execution time and memory usage.

Chapter 5 presents an optimization technique, called *Object Ownership* for RTRC. Even though it is designed for RTRC it can be used to optimize all incremental garbage collectors. A benchmark of a version of the technique is also presented.

Chapter 6 presents the design of the *static garbage collection* technique. The chapter also contains a discussion of some shortcomings of the technique and how to solve these problems.

Chapter 7 presents implementations of the techniques presented in the thesis in three different runtime systems. The first implementation is of RTRC in C. Some manual work is needed to use it, but it gives the developer direct control of the generated code. Then follows a presentation of the RTRC implementation in the CoSy framework, which is an industrial strength compiler framework. The third implementation is that of the static garbage collector in the Jamaica VM. The Jamaica VM is a commercially available real-time Java implementation. It is currently the only Java implementation with a runtime system that has been published and proved predictable.

Chapter 8 presents measurements of the C and CoSy implementations of RTRC and discusses the results.

Chapter 9 presents related work.

Chapter 10 presents some ideas of future direction of this work.

Chapter 11 concludes the work presented in this thesis.

1.5 Publications

This thesis is partially based on publications of the RTRC listed below. The Object Ownership optimization and the static garbage collector have not yet been published. However, we plan to submit articles that present the techniques to conferences and journals.

[Rit99c] Tobias Ritzau. *Real Time Reference Counting in RT-Java*. Licentiate thesis, Linköping University, March 1999.

[RBLP00] Tobias Ritzau, Marcel Beemster, Florian Liekweg, and Christian Probst. JoC — the JOSES compiler. Presented at the Java for Embedded Systems Workshop, London, May 2000.

[Rit01] Tobias Ritzau. Hard real-time reference counting without external fragmentation. In Dr. Uwe Assmann, editor, *Java Optimization Strategies for Embedded Systems Workshop at ETAPS 2001*, Genova, Italy, April 2001.

[RF02] Tobias Ritzau and Peter Fritzson. Decreasing memory overhead in hard real-time garbage collection. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, volume 2491 of *Lecture Notes in Computer Science*, pages 213–226. Springer, 2002.

We also consider the possibility of integrating RTRC with the JDrums technique as presented in the following publications. JDrums is a framework for dynamic updating of executing Java applications.

[ACR98] Jesper Andersson, Marcus Comstedt, and Tobias Ritzau. Runtime support for dynamic Java architectures. In *Proceedings of the Workshop on Object-Oriented Software Architectures*. The ECOOP'98 Workshop on Object-Oriented Software Architectures. Brussels, July 1998.

[AR00] Jesper Andersson and Tobias Ritzau. Dynamic code update in JDrums. In *Procedings of the First workshop on Software Engineering for Wearable and Pervasive Computing (SEWPC) in Conjunction with ICSE'2000*, Limerick, June 2000.

— A wizard is never late, Frodo Baggins, nor is he early.
He arrives precisely when he means to.

Gandalf

Chapter 2

Real-Time Systems

This chapter gives a brief overview of real-time systems. It is presented here to give the reader an insight in the systems we are targeting, not to give a full introduction to the topic. Burns and Wellings [BW89] give a good starting point for deeper study of this topic.

2.1 Definition

There are many interpretations of the term *real-time system*. The Oxford Dictionary of Computing gives the following definition:

> *Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.*

Burns and Wellings [BW89] are more concise when they state:

> *The correctness of a real-time system depends not only on the logical results of the computation, but also on the time at which the results are produced.*

All agree that time is an important factor and that the system must respond within a specified time, a *deadline*. The consequence of missing a deadline might not be critical in some systems, but in other it might be disastrous. It is common to model a real-time system by dividing it into separate *tasks*. These tasks can, for example, be implemented using processes, threads or co-routines.

2.2 Categorizing Real-Time Systems

The terms *soft* and *hard real-time systems* are often used to categorize real-time systems. Another category, interactive systems, is closely related to soft real-time systems. The borders between these categories are very unclear, and it is not always easy to decide to what category a system belongs.

2.2.1 Interactive Systems

Interactive systems have an active dialog with their users. Normally the users of an interactive system want response as soon as possible, but it is also important to keep the variance of the response limited. If an interactive system normally responds within tenths of a second, and in some cases after a few seconds, it could lead to repeated input from the user. An example of systems having problem with this are the early digital satellite receivers which sometime have a terrible response time that often lead to repeated input. This is extremely annoying when the input toggles a feature. However, most systems are so fast that the variance is not a problem.

Other examples of interactive systems are calculators, database frontends, and the editor in which this thesis is written. Most of these systems do not have any specified deadlines and no analysis is performed to guarantee response time. In the worst case a slow response lead to annoyance (which on the other hand could render the system useless.)

2.2.2 Soft Real-Time

A soft real-time system has specified deadlines, but an occasional slightly missed deadline does not lead to disaster. However, the quality of the result is reduced. Multi-media systems, e.g. audio and video decoders, are examples of soft real-time systems. At worst a single missed deadline would cause a hardly noticeable jerk in the audio or video stream. This does not mean that deadlines can be ignored, since no user would like to use a decoder that produces jerky output.

Another example is a freezer. An occasionally slightly missed deadline does not cause any problems or is easy to fix afterward. However, if misses are frequent or large it causes the system to malfunction, which in turn could cause the food in the freezer to go bad. Which indeed is a disaster if that is the only food you have.

2.2.3 Hard Real-Time

A hard real-time system has strict deadlines that should be guaranteed to be met at all times. Even an occasional slightly missed deadline in a hard real-time system could lead to a disaster, e.g. fatalities or large financial

losses. Examples of hard real-time systems are airplane flight controllers and medical equipment. These systems must, in all cases, meet their deadlines.

However, many hard real-time systems can cope with an occasional slightly missed deadline, i.e. a missed deadline is not necessarily fatal. A missed deadline in an airplane control system could send the plane off course for a moment, which is not a problem during mid-flight, but if it happens during the landing of a plane, it might crash.

Another example of a hard real-time system is a paper mill controller. The paper should be stretched at all times, and if a missed deadline causes too much tension, the paper might tear which causes a lot of extra work and thus financial losses.

2.3 Predictability

To be able to meet a deadline the system must be predictable, i.e. it must be possible to foresee the runtime behavior of the system for all possible inputs.

Some literature [BW89] goes to an extreme and prohibits the use of certain language features. These include:

- recursion
- dynamic allocation of memory
- dynamic creation of processes

This is too restrictive for others. Generally, the features listed above might be unpredictable, but they can be implemented to be predictable and then they can also be used. It is customary that loops are given an explicit maximum loop count. The same can be applied to recursion to make it as predictable.

Under normal circumstances, dynamically allocating memory would interfere with the virtual memory manager, which has terrible worst case execution time. However, using virtual memory in real-time system might not be a good idea anyway. Dynamically allocating memory can be implemented in a fully predictable manner as described in Chapter 4. As an alternate work-around a simple memory manager can be implemented by keeping a pool of memory regions that can be allocated predictably by the system.

Creating new processes need not be a problem. However, scheduling the new process might be. An alternate solution would be to use a pool of idle processes which are scheduled using specified parameters, e.g. with preset frequencies.

An important property of a real-time system is that its resource usage must be predictable in a way that the system does not fail because of resource shortage. Two important resources are execution time and memory. These are discussed in the following sections.

2.3.1 Execution Time

There must be an upper bound of the execution time of all tasks regardless of input. This bound is called *worst case execution time* (WCET). To be able to calculate an upper bound, all possible execution paths need to be considered. Certain programming features can cause infinite execution paths, and should be used with caution. These include while-loops and recursion. It is common to limit the number of iterations in loops and the depth of recursions. Recursion can also cause the stack to overflow, so extra caution is needed if it is used.

Virtual method calls can cause very pessimistic WCETs if methods are analyzed separately. If a method is redefined in several subclasses and there is no knowledge of the exact types of the objects, all methods need to be taken in consideration. Inter-class analysis is needed to limit the possible types of objects. One such approach is rapid type analysis [Bac97]. The same problem may occur in all selection statements. Common solutions include using constant propagation and interval analysis.

2.3.2 Memory Usage

Since all possible execution paths are needed to calculate the WCET, it is also known how memory is allocated. However, the memory analysis is more complicated since the size of memory regions need not be constant, and because of internal and external fragmentation. External fragmentation occurs when no contiguous memory region is large enough to fit a new object, but the total amount of memory is larger than the object. Internal fragmentation occurs when more memory than requested is allocated, which causes unused memory in the in the end of the allocated memory. If a garbage collector is used, the problem is even more difficult since the instructions that reclaim memory are not explicit. Thus, it is hard to tell when the garbage collector will be able to reclaim memory.

The amount of memory allocated at every allocation must be limited. Constant propagation and interval analysis are useful if the size is not constant. The fragmentation problem is more serious. Depending on the allocation strategy, the internal fragmentation may be predictable (but maybe not constant.) However, most allocators give unpredictable external fragmentation, i.e. no large enough contiguous memory region may be available even though the total amount of free memory is sufficient. A system that must not fail can not suffer from external fragmentation. External

fragmentation can be avoided, for example using compaction and fixed allocation units.

It is not enough to guarantee fully predictable allocation. Deallocation must also be predictable. That problem is strongly connected to the problems of allocation, but if automatic memory management, or garbage collection, is to be supported the problems increase. Not only must the execution time of the garbage collector be predictable, it must also be guaranteed that memory is reclaimed at a pace that guarantees that the system never runs out of memory.

One might question the use of garbage collection in a real-time system. On the other hand garbage collection relieves programmers of the difficult task of finding positions in the code where memory can be safely reclaimed without leaving memory leaks. However, in a real-time system the programmers need to have full control, therefore it should be clear at what positions memory should be reclaimed. Also, if there is a memory related programming error in the software, a garbage collector might indicate it by some missed deadlines. A system without a garbage collector could fail completely during the same circumstances.

2.4 Scheduling

A real-time system consists of a set of tasks. A task is either periodic or aperiodic. A *periodic task* has explicit deadlines and typically samples data or executes a control loop. An *aperiodic task* is activated by an external (asynchronous) event. An aperiodic task must respond within a specified *response time*.

The notion of *temporal scopes* [LG85], can be used to describe the timing specification of a real-time application. The attributes of a temporal scope are illustrated in Figure 2.1, and include:

- Deadline
- Minimum delay
- Maximum delay
- Maximum execution time
- Maximum elapse time

To simplify the discussion we will use the term *deadline* for all timing constraints. A *schedule* is an ordering of tasks, which may be split into subtasks. A real-time system is said to be *schedulable* if there exists a schedule such that all tasks will meet all upcoming deadlines.

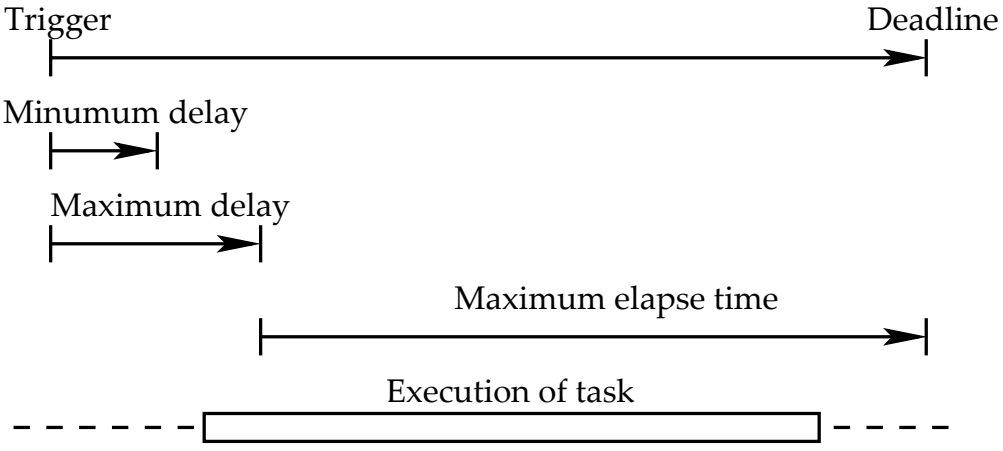


Figure 2.1. Temporal scopes

2.4.1 Cyclic Executive

The most primitive form of scheduling a set of tasks is to do it statically using a technique called cyclic executive. It is often required to split a task into parts to make the application schedulable. An example is presented below. If the number of processes is small, this approach is feasible, but the problem gets drastically more complex when the number of tasks increases. If the system contains aperiodic tasks, these must be polled.

Example – Cyclic Executive

Tasks A and B should be scheduled according to the following specification: Task A has an execution time of 0.2 s and is periodic and should be invoked twice every second. Task B has an execution time of 0.1 s and should be invoked 5 times per second.

B	A_0	B	A_1	B	$-$	B	A_0	B	A_1
0	1	2	3	4	5	6	7	8	9

This solution requires a resolution of 10 units per second. Task A needs to be split into two tasks, and one time slot is left unused. This sequence can be repeated indefinitely, and all deadlines will be met. Thus, task A and B are schedulable.

2.4.2 Pre-emptive Priority Scheduling

The idea is simple: assign a priority to every task, and execute the task with the highest priority among the tasks that are ready to execute. Pre-emptive means that a task can be interrupted by other tasks, e.g. a high priority process can interrupt a low priority process when the high priority

process becomes ready to execute. Most modern (workstation and server) operating systems use some kind of pre-emptive priority scheduling (often in combination with other strategies.) Aperiodic tasks cause no problem for the scheduler, but it makes schedulability analysis more complex.

One problem that can arise is *priority inversion*, which is best explained in an example. Consider three tasks, A, B and C. Task A has highest priority followed by task B, and task C has the lowest priority. First task C acquires a mutually exclusive resource. Task A interrupts C and tries to acquire the same resource. Task A has to wait for C to be finished, and while C still uses the resource task B interrupts C and starts executing. Now task A has to wait for B to finish even though A has higher priority and they do not share any resources. This is called priority inversion. A solution is to let C inherit the priority of A, as soon as A tries to acquire the shared resource. Then B can not interrupt C, and A does not need to wait for B. This technique is called *priority inheritance*.

It is hard to statically predict how long a task can be blocked in the general case. Using priority inheritance this can be achieved, but the prediction is usually too pessimistic. The inheritance protocol does not prohibit deadlocks either. These problems are solved by the *ceiling protocol* [SRL88]. The ceiling protocol guarantees that no deadlocks occur, that chains of blocks can not occur, and that a high priority task can only be blocked once by a lower priority task per activation. The ceiling protocol has evolved into the *optimal mutex policy* [RSLR95], which further guarantees that no other priority inheritance policy can guarantee a better worst case blocking duration.

The priorities of the tasks can be assigned statically or dynamically. Some approaches to assigning priorities are presented in the following sections.

Rate Monotonic Scheduling

A set of independent tasks can be scheduled using rate monotonic scheduling [CJ73]. The priorities are assigned according to the frequencies of the tasks. Highest frequency gets highest priority. If the tasks are independent and the overall CPU utilization is below 69 %, rate monotonic scheduling will guarantee that all deadlines are met indefinitely. Regrettably, there are inter-task dependencies in most real-time systems, but in combination with for example the ceiling protocol a predictable solution can be achieved.

Another problem is how to handle aperiodic tasks. Since rate monotonic scheduling is a static scheduling technique, aperiodic tasks have to be handled as periodic tasks with a frequency. This could lead to very pessimistic predictions. Still, rate monotonic scheduling is a good start when assigning priorities to tasks.

Earliest Deadline First

Earliest deadline first is a dynamic approach where the scheduler must have knowledge of all tasks deadlines. The task with the earliest deadline is allowed to execute first. This technique can handle aperiodic tasks as well as periodic ones. It can also be proven that a system that can be scheduled using rate monotonic scheduling can also be scheduled using earliest deadline first.

Least Slack Time First

Least slack time first is also a dynamic technique. The scheduler needs information about all deadlines and of the execution time of all tasks. The task with the least time left to its deadline after its execution is given control. This scheduler can schedule all set of tasks that a rate monotonic scheduler can handle, and can also handle aperiodic tasks well. The advantage of this scheduler compared to earliest deadline first is that if a task takes longer time than expected, there is a better chance that all deadlines will be met. The disadvantage is that the scheduler needs more information, but that information is normally available in a real-time system.

2.5 Which Systems Are Hard?

As stated above, the border between soft and hard real-time systems is fuzzy. It is easy to separate them by saying that in hard real-time systems a deadline must never be missed and in soft real-time systems an occasional miss may occur. The problem becomes to decide if an occasional deadline can be missed or not. Most real-time systems can cope with a slightly missed deadline, but that does not make all real-time systems soft.

When designing a real-time system one have to specify the deadlines of the different tasks in the system. In some systems, e.g. audio and video decoders, it is easy, since there is a specified frequency in the output stream that has to be kept, but in other systems like cruise controls it is much harder. By decreasing the frequency of the controller and give its task longer time to complete its execution, the system need less computing power but will act in a jerky way. The decision is often based on a combination of calculations and testing. Therefore a slightly missed deadline need not cause a system failure, even in hard real-time systems. However, a miss might still be disastrous, therefore a hard real-time system must be designed with that in mind.

— The time you enjoy wasting is not wasted time.

Bertrand Russel

Chapter 3

Garbage Collection Techniques

This chapter gives an overview of fundamental garbage collection techniques. These techniques are commonly used as the basis of new garbage collectors. In this chapter we also discuss what the requirements of a real-time garbage collector are, and the chapter concludes with a discussion of which technique to use as the basis of our real-time garbage collector.

The purpose of a garbage collector is to reclaim memory regions that will not be accessed in the future. Most techniques do this by reclaiming memory that can not be referenced anymore, i.e. they reclaim memory regions that are not referenced by any reference in the system.

By handing over the responsibility of freeing up memory to the runtime system, many memory related programming errors are eliminated. These errors are otherwise very hard to detect and correct, since the code where the error is detected may have very little to do with the code that generates the error. Typical errors are memory leaks, premature freeing, and multiple freeing of the same region.

Memory leaks occur when allocated memory is forgotten and is not reclaimed. A long running application with a memory leak will eventually run the system out of memory.

If a memory region is prematurely freed, i.e. reclaimed while it is still in use, the data in the region will be corrupted by multiple uses. This can corrupt the internal data structure of the memory manager, which may cause obscure errors later. Even if the memory manager is not corrupted, the errors can be very hard to detect. Best is probably if it causes the system to crash, since it may be much harder to find out why the output is wrong. Symptoms from double usage may be very confusing, e.g. if the type information is changed in an object-oriented system, wrong methods may be called.

Finally, if a region is reclaimed more than once, the internal data structure of the memory manager may be corrupted. This can cause many obscure errors, e.g. a crash when the memory region is later allocated again. All these problems can be eliminated by using a garbage collector.

3.1 Terminology

object *object graph* *child* *parent*

To make the presentation simpler everything that is allocated into memory regions is called *objects*. These objects are connected via references that form the directed edges of the *object graph* in which the objects are nodes. The objects which are directly referenced from an object, A, are called *children* of A, and if A is a child of B, then B is a *parent* of A. An object can have many parents.

mutator *collector*

An application using garbage collection can be divided into a *mutator* and a *collector*. The mutator is the actual application, which mutates the object graph, hence the name. The rest of the application is called the collector and performs the garbage collection work.

root

Objects that always exist in a system are called *roots*. Roots consist of permanent objects (e.g. global and static objects), run-time stacks, and processor registers.

reachable *live*

An object is *reachable* if there exists a path from one of the root objects to the object in the object graph, and objects that will be used by the mutator in the future are called *live*. Note that a reachable object does not have to be live, but all live objects are reachable. To simplify garbage collectors, they often assume that all reachable objects are live. This is a safe assumption, but increases the memory usage of applications.

internal fragmentation *external fragmentation*

Fragmentation can appear in two versions. *Internal fragmentation* appears when more memory than requested is allocated. Thus, some memory in the end of the allocated region remains unused. *External fragmentation*, on the other hand, appears when no contiguous memory region is large enough, even though the total amount of unused memory is.

moving *non-moving* *compact*

Some garbage collection techniques allow objects to be moved around in memory. These techniques are denoted *moving* garbage collectors. Techniques that do not move objects are called *non-moving*. The purpose of a moving collector can be to improve cache performance, or to *compact* live objects, i.e. to move them into a contiguous memory range. Compaction eliminates external fragmentation, and since all free memory becomes contiguous, allocation becomes a very cheap operation of incrementing a pointer in the free region.

GC cycle *incremental*

If a garbage collector performs a complete *garbage collection cycle*, i.e. finds dead objects and reclaims them each time it is invoked, long unpredictable pauses may delay the mutator. An *incremental garbage collector* can be interrupted so that pause times of the mutator can be shorter. Garbage

collectors that can not be interrupted are called *stop-the-world garbage collectors*. *stop-the-world*

While performing garbage collection work, objects can be divided into three groups. This classification is called *tricolor marking* and was introduced by Dijkstra et al. [DLM+78]. The objects which have been processed by the collector are *black*. This means that all children of black objects have been found. Objects which have been found to be reachable, but have not yet been fully processed, i.e. all their children may not have been found, are colored *grey*. The remaining objects have not yet been reached by the collector; these are colored *white*. If an object is white when all reachable objects have been found, i.e. no grey objects exist, it can not be live. Thus, it can safely be reclaimed by the collector. *tricolor marking* *black* *grey* *white*

If an incremental garbage collector is used, the object graph is mutated while it is being analyzed by the collector. If a reference to a white object is stored in a black object, it may be missed by the collector. To ensure that this does not occur, the mutator is responsible of keeping white objects from being referenced by black ones. This is done by inserting code *barriers* that protect the mutations of the object graph. Two kinds of barriers exist: read- and write-barriers. Depending on the garbage collection technique only one of them may be required. *barriers*

A *read-barrier* is used to protect reference read operations so the mutator never sees a white object (and can thus never store references to them.) If a white object is accessed, it has to be immediately colored grey or black. A *write-barrier*, on the other hand, protects reference store operations. If a reference to a white object is stored in a black object, the black object could be degraded to grey, or the white object can be colored grey (or black.) *read-barrier* *write-barrier*

A *snapshot-at-the-beginning collector* only reclaims memory that was garbage at the start of the GC cycle. Thus, at the start of a new cycle a snapshot is taken. All objects that are dead in the snapshot will be reclaimed at the end of the cycle. A consequence of this technique is that garbage will float from one cycle to the next where in most cases it is reclaimed. The opposite of snapshot-at-the-beginning collectors are the *incremental-update collectors* that keeps the garbage collector up to date at all times. The garbage collector tries to reclaim all garbage at the end of each cycle. An incremental-update-collector can leave some garbage floating between cycles. For example, many algorithms do not reclaim objects that are allocated during the current cycle. *snapshot-at-the-beginning* *incremental-update*

If garbage is allowed to float between cycles, it may be necessary to complete more than one cycle to free sufficient amounts of memory. This may lead to longer pauses for the mutator. However this is very rare and many applications can cope with such interruptions.

All garbage collectors have to be able to analyze the object graph. *Conservative garbage collectors* use heuristics to find out whether a memory cell represents a reference or other data. The heuristics must find all references, *conservative*

exact

but may also regard other data as references. *Exact garbage collectors*, on the other hand, have exact knowledge of the types at run-time, and can thus use this information to find the children of every object. Some techniques can be implemented as either conservative or exact. However, real-time garbage collectors must be exact, otherwise they can not be predictable. In the pseudo code that follows, it is assumed that an iterator of the children of an object is returned using the `getChildren()` method.

3.2 Reference Counting

Reference counting differs radically from other garbage collection techniques. Instead of having a separate collector, the collection process is interleaved with the mutator using a code similar to a write-barrier. It is not really a write-barrier, but it is used in the same way. To make comparisons easier, the routine will be called a write-barrier in this thesis.

The idea of reference counting is to count the number of references to each object. When the reference count falls to zero, there are no more references to the object. Thus, the object becomes unreachable and can be reclaimed. A problem that may arise is an overflow in the reference counter. An overflow would cause the reference counter to be corrupted, e.g. become zero, which probably causes the system to fail. There are two main approaches to eliminate this problem. Overflows can be allowed by locking the counter at a position. The real value of the reference counter can later be calculated by following all references and counting the references to objects with locked reference counts. There are reference counters that only use one bit for the reference counter to take advantage of the fact that in some systems many objects only have one reference. The other solution is to use a reference counter that is too large to overflow, e.g. a 32 bit reference counter in most current systems are more than enough since their memory can not hold 2^{32} references.

Each object must be associated with a reference count. This reference count is updated when the number of references to the object is changed. This happens each time a reference is assigned to a variable, when a reference is passed as an argument to a method, when a variable goes out of scope, and when a reference is returned from a method. Reference assignment is shown in Figure 3.1. Passing references as arguments to a method increases the reference count by one. When a method returns, all local references must be released (see Figure 3.1.) If a reference is returned from a method, the referenced object's reference count must be increased before the method returns, and decreased when the return value has been passed on to the caller of the method.

Allocation and free operations can be performed using any technique used for manual memory management. The allocation techniques can be chosen and tuned according to the requirements of the system.

```
algorithm release(Object obj)
    obj.refCount ← obj.refCount - 1
    if obj.refCount = 0 then
        foreach child in obj.getChildren() do
            if child ≠ null then
                release(child)
            end if
        end loop
        free(obj)
    end if
end

algorithm write-barrier(Object in out lhs, Object rhs)
    { Maintain the reference counts of lhs and rhs }
    if rhs ≠ null then
        rhs.refCount ← rhs.refCount + 1
    end if
    if lhs ≠ null then
        release(lhs)
    end if
    lhs ← rhs
end
```

Figure 3.1. Standard reference counting

It is very important to increment the reference count of the right-hand side before the reference count of the left-hand side is decreased, otherwise the following code would cause a serious program fault.

```
Foo a ← new Foo()
{ The reference count of a is 1 }
a ← a
{ If the reference count is first decremented it will reach zero! }
```

Even though this code looks bizarre, it is correct. The problem can also be solved by comparing `lhs` to `rhs` in the write barrier. If they are equal, the write-barrier does nothing. This would improve the execution time of assignments like the one above, but slow down other assignments. While the special assignments should be very rare, it is better to do the incrementing first and get rid of the problem.

If reference counting is to be used in a multi-threaded system, some synchronization has to be performed. Problems will arise if a thread is interrupted in the middle of an increment or decrement operation of a reference count while another thread is updating the same reference count. To ensure that this never happens, the counter update operations must be performed atomically.

The allocation and free operations will probably need locks as well. This is no different from the synchronization in other multi-threaded memory allocators.

3.2.1 Lazy Reference Counting

Weizenbaum [Wei63] describes a technique to eliminate the recursive behavior of the decrement operation. The technique eliminates recursion by adding objects to a list when their reference count falls to zero, instead of decrementing the reference counts of their children. This list is called a to-be-free list, since the objects of the list may still refer to live objects. When a new object is needed from the to-be-free list, it is removed and then the reference counts of its children is decremented, which may add new objects to the to-be-free list. Thus, it is important that the child references are not corrupted when objects are added to the list. The to-be-free list can be implemented as a linked list, and the reference count field can be used to store the links, since the reference count is always zero when objects are in the list. The algorithm is presented in Figure 3.2.

However, this technique can not be used directly in systems where objects have varying sizes. If differently sized objects are used, the worst case of allocation becomes to free all objects on the heap (just to find an object of right size.)

3.2.2 Cyclic Reference Counting

A major disadvantage of reference counting is its inability to reclaim dead cyclic data structures. If a cyclic data structure becomes unreachable from the roots of the system, the objects in it keep each others reference count larger than zero. Thus, they are not reclaimed.

One approach to collect dead cyclic data structures, first proposed by Christopher [Chr84], is to find potentially cyclic data structures and investigate whether they are isolated cycles or not. The basic idea is to investigate objects whose reference counts are decremented to a value greater than zero. These objects could be part of isolated (dead) cycles and are further investigated. During the investigation, the objects child references are decremented recursively. If all traversed objects have a reference count of zero, there are no external references. Thus, an isolated cyclic data structure has been found and can be reclaimed. If any object has a reference count that is greater or equal to one, there is at least one external reference and all reference counts need to be restored. This is costly, and many improvements have been proposed.

Bacon and Rajan [BR01] proposes an improvement of Christopher's technique. This technique compares well to other garbage collectors. Unfortunately, the WCET is too large for hard real-time systems. The main

```
algorithm new(int size) returns Object
    { Allocate without distorting memory (from tbf-list) }
    Object obj ← allocate(size)
    foreach child in obj.getChildren() do
        if child ≠ null then
            release(child)
        end if
    end loop

    { Call the constructor }
    obj.init()
    return ref
end

algorithm release(Object obj)
    obj.refCount ← obj.refCount - 1
    if obj.refCount = 0 then
        { Free the object without distorting the memory }
        { (add to tbf-list) }
        free(obj)
    end if
end
```

Figure 3.2. Lazy reference counting

idea is to check several sub-graphs at once, instead of doing a cycle check each time a reference count is decremented to something greater than zero. Herein lies the WCET problem, since all objects on the heap may need to be investigated. Bacon and Rajan also proposes other improvements such as marking objects that can not possibly be part of any cyclic data structures, so these need not be checked during run-time.

We propose yet another optimization that may be successful. If all objects are marked with a time stamp, this could be checked while decreasing reference counts. The idea is that non-cyclic references should go from a younger object to older ones. If this property holds, no cycle could be created if a new edge (reference) is created from a young object to an older one. On the other hand if a younger object is referenced by an older one, the reference can be part of a cycle. If an edge (reference) from an older object to a younger one does not create a cycle, the younger object and the objects that are reachable from it need to be aged to uphold the property as described above.

This optimization will reduce the need for doing cycle checks, but it also introduces the aging process and an extra field to store the age of an object. A problem occurs if the age of an object overflows. This problem remains to be solved.

```
algorithm write-barrier(Object lhsParent, Object in out lhs, Object rhs)
    { Only count inter-group references }
    Group g ← lhsParent.group
    if rhs ≠ null and then g ≠ rhs.group then
        rhs.group.refCount ← rhs.group.refCount + 1
    end if
    if lhs ≠ null and then g ≠ lhs.group then
        lhs.group.refCount ← lhs.group.refCount - 1
        if lhs.group.refCount = 0 then
            freeGroup(lhs.group)
        end if
    end if
    lhs ← rhs
end
```

Figure 3.3. Reference counting using Bobrow's groups

Depending on the run-time behavior of the systems, it might be better to reverse the property and only allow references from older to younger objects. In any case, the excessive WCET makes the solution inappropriate for hard real-time systems.

3.2.3 Bobrow's Approach to Reclaim Cycles

Using Bobrow's technique, as presented in Figure 3.3, objects are divided into groups that may contain cycles. The groups must be formed so that no cycle involves more than one group. All groups maintain a reference count of references from other groups (external references). An example is presented in Figure 3.4. External references are not counted in individual objects. All objects in a group can be reclaimed when the reference count of a group becomes zero. Thus, cycles are collected.

The disadvantage of Bobrow's technique is that it is difficult to assign groups to objects. One must first decide how to divide all objects into groups, and then it must be possible to tell the runtime system to which group recently allocated objects belong. The latter problem is minor when all code is written with the technique in mind. However, reusing legacy code can cause major problem.

A major problem occurs when an object is part of more than one cycle. Since no cycle can be part of more than one group, all cycles that share a node must be in the same group. If a cycle becomes isolated within a group it is still not reclaimed until the group is, neither is any other object that is (or has been) part of the group. Thus, Bobrow's groups do not completely solve the problem. However, it can be useful in many cases.

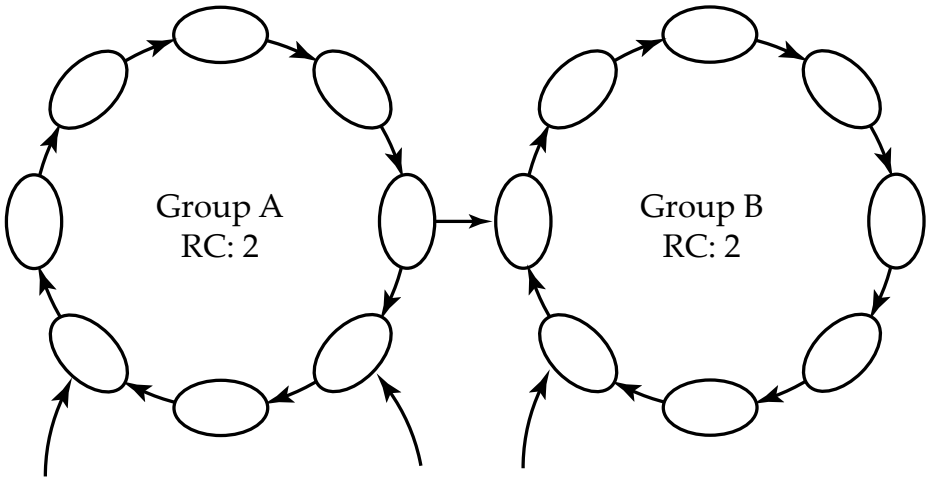


Figure 3.4. Two groups in an object graph using Bobrow's cycles. The group A has a reference count of two. So does group B, since the reference from group A is external to group B.

3.2.4 Deferred Reference Counting

Deutch and Bobrow [DB76] discovered that much time was spent updating reference counts due to local variables, i.e. variables stored on the stack. To improve the performance, Deutch and Bobrow proposed that local variables should not update reference counts. Thus, a reference count of zero does not mean that an object can be reclaimed, only that it is not accessed from the heap. The decrement operation is modified to add objects with a reference count of zero to a table called Zero-Count-Table (ZCT). This table has to be fast and is commonly implemented as a hash table or a bitmap. Since the decrement operation does not reclaim memory, a separate routine must be used for that. This routine scans the stack and increment the reference counts of all objects that are referred from the stack. Then, the ZCT is traversed to find objects with a reference count of zero. These objects are neither referred from the stack nor from the heap, and can be reclaimed as before. Finally the reference counts of objects referred from the stack need to be reset by decrementing them again. The algorithms are presented in Figure 3.5.

Unfortunately, deferred reference counting is not suitable for hard real-time. First, objects are not reclaimed immediately when they become unreachable. Thus, memory usage is increased. And second, the routine that collects garbage can not be interrupted, which is unacceptable in most hard real-time systems.

```
algorithm decrement(Object obj)
    obj.refCount ← obj.refCount - 1
    if obj.refCount = 0 then
        zct.add(obj)
    end if
end

algorithm increment(Object obj)
    obj.refCount ← obj.refCount + 1
    zct.remove(obj)
end

algorithm collect()
    foreach obj in the stack do
        increment(obj)
    end loop
    foreach obj in zct do
        if obj.refCount = 0 then
            foreach child in obj.getChildren() do
                decrement(child)
            end loop
            free(obj)
        end if
    end loop
    foreach obj in the stack do
        decrement(obj)
    end loop
end
```

Figure 3.5. Deferred reference counting

3.3 Mark-and-Sweep

Mark-and-sweep collectors perform the garbage collection in two phases. First, live memory is marked by traversing the object graph starting at the roots. Next all unmarked memory is reclaimed in the sweep phase. The algorithm starts by marking the roots. Marking an object includes finding its children and marking them. By marking the roots, all reachable objects will be marked. The sweeping phase traverses the heap and all unmarked objects found are reclaimed.

It is a common choice to start the garbage collector from the memory allocation function. In a non-incremental algorithm the collector is often started when the system runs out of memory. When using incremental collectors, some work is commonly performed every time memory is allocated. The amount of work in each increment is often proportional to the amount of allocated memory.

3.3.1 Incremental Mark-and-Sweep

The classical implementation of the mark phase is recursive. However, that is not suitable if the collector is to make small increments at each invocation. A frequent solution is to use a mark stack where all grey objects are stored. When using a mark stack, only one bit is required to represent the color of an object. All objects on the mark stack are grey. Objects which are not on the mark stack are white if the mark bit is unset, otherwise they are black.

Numerous incremental mark-and-sweep garbage collectors have been proposed. In this section two algorithms are briefly described. The first algorithm is a snapshot-at-the-beginning-collector by Yuasa [Yua90], the other is an incremental update algorithm by Dijkstra et al. [DLM+76].

3.3.2 Yuasa's Algorithm

Since Yuasa's algorithm is a snapshot-at-the-beginning algorithm, it is the most conservative of these algorithms. The algorithm works in one of three phases: idle, marking, or sweeping. The algorithm starts out in the idle phase. Each time an object is allocated, the amount of free memory is checked against a threshold value. If the amount of free memory is lower than the threshold, the collector goes into the marking phase, marks all roots except the stack as grey, and makes a copy of the runtime stack. The stack is copied to get a snapshot of the state at the beginning of the GC cycle.

During the marking phase, a write-barrier ensures that the object referenced by the left-hand side of the assignment is colored grey if it was previously white (see Figure 3.6 and 3.7). When an object is allocated during the marking phase, the collector starts traversing the object graph. The traversal finishes when a fixed number of objects have been processed or when the mark stack is empty. If the mark stack is empty, objects from the copied program stack are pushed onto it. The number of transferred objects is limited by a constant. If the copied stack also becomes empty, the collector goes into its sweeping phase.

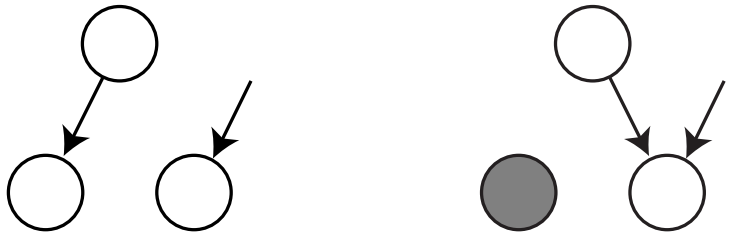
Allocations in the sweeping phase are preceded by a sweep of a constant number of white objects. When all white objects have been swept, the collector goes back into its idle phase.

Except for the write-barrier, all work in Yuasa's collector is done while allocating new objects. When the collector is invoked, it always processes a fixed number of objects. Consequently, Yuasa describes this algorithm as real-time even though no guarantee of memory availability is given. The only guarantee that is given is the WCET of the barrier.

Even though this is a snapshot-at-the-beginning algorithm, new objects are not necessarily marked black. During the idle phase all objects are

```
algorithm write-barrier(Object in out lhs, Object rhs)
    { Color lhs grey if it is white }
    if phase = marking then
        if lhs.isWhite() then
            gcStack.push(lhs)
        end if
    end if
    lhs ← rhs
end
```

Figure 3.6. The write barrier in Yuasa's algorithm



(a) Before (b) After

Figure 3.7. Yuasa's write-barrier

marked white, during the mark phase they are marked black, and all objects allocated "behind" the sweeping front are allocated white when the collector is sweeping, the ones that risk being swept are allocated black.

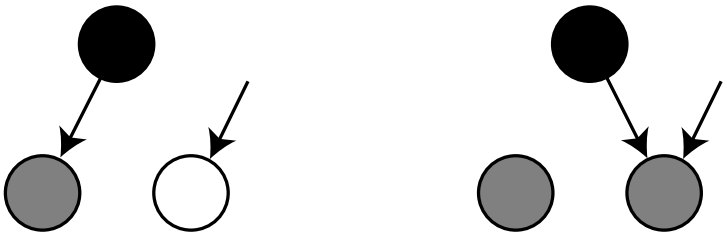
The write-barrier is special since it permits a white object to be referenced from black objects, but the implementation guarantees that the white object is referenced from at least one other reachable object, otherwise it could not be referenced in the assignment. The object referring to the white object must be non-black, otherwise the white objects would have been non-white. Thus, the white object will be found by the collector if it is reachable at the end of the mark phase.

3.3.3 Dijkstra's Algorithm

Dijkstra et al. has presented an incremental-update garbage collector based mark-and-sweep [DLM$^+$76]. The algorithm is designed to be simple to prove correct. Even though it is an incremental-update-collector, it is quite conservative. The write-barrier, shown in Figure 3.8 and 3.9, shades the right-hand side of the assignment to grey if it was previously white. This ensures that there can be no reference from a black object to a white one.

```
algorithm write-barrier(Object out lhs, Object rhs)
    { Color rhs grey is it is white }
    lhs ← rhs
    if rhs.isWhite() then
        rhs.setColor(grey)
    end if
end
```

Figure 3.8. Dijkstra's incremental update write barrier



(a) Before (b) After

Figure 3.9. Dijkstra's write-barrier

To simplify the proof of this algorithm the free-list is considered to be reachable and should also be marked. This implies that all new objects are allocated as grey or black, depending on the color of the head of the free list. Thus, a new object survives at least one cycle, even if it becomes unreachable before the end of it.

To determine when the marking phase is over, the run-time stack is scanned for grey objects. If a grey object is found, the marking starts from that object. This gives a worst-case execution time which is quadratic to the size of the heap. Kung and Song [KS77] have improved the performance of the algorithm using auxiliary data structures.

3.4 Mark-and-Compact

If fragmentation is a problem, a solution is to compact the heap, i.e. to move objects so that free memory becomes contiguous. Since the heap is compacted, the sweep phase is superfluous.

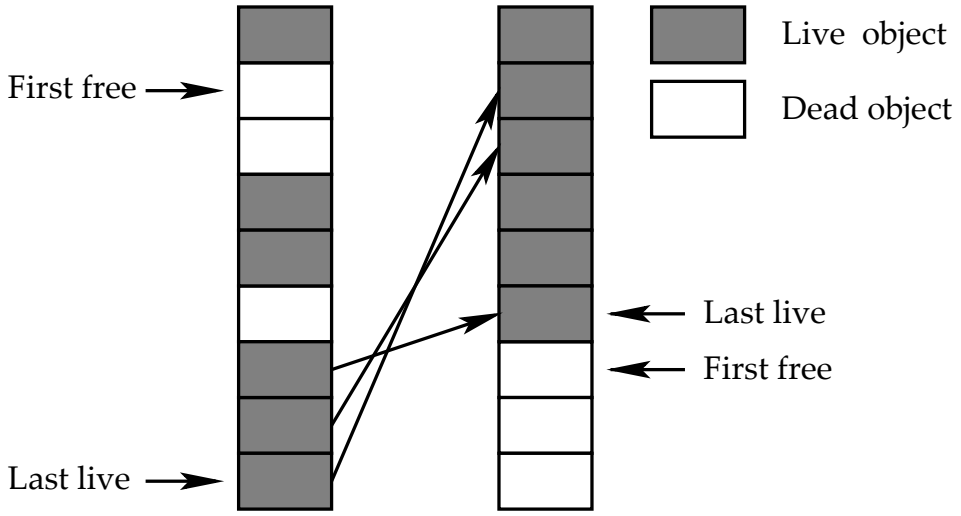


Figure 3.10. The two-finger algorithm

3.4.1 Compaction Methods

Compacting the heap includes moving live regions and updating pointers to the regions which have been moved. Several mark-and-compact techniques have been proposed.

Two-Finger Algorithm

This algorithm, described by Saunders [Sau74], is designed for equal sized objects. Two pointers are used to point out the first free region, and the last reachable region (see Figure 3.10). Objects are moved from the end of the heap to the first free slot. When the two pointers meet, all memory is compacted. Forward references, which are stored in the first cell of moved objects, are then used to update references from other objects. The algorithm does not require any extra space (except for the mark bit), but the objects are re-ordered. This is not a problem in most systems, but some systems depend on the ordering of objects.

Forwarding-Address Algorithm

These algorithms can keep variable sized objects in order. An extra field is used to store a forward reference in each object. Different strategies can be used to compact live regions. In some cases it is important to keep the objects in order, but other strategies can be used to save time.

The drawback of these algorithms is the space overhead and an extra phase to compute the new addresses of objects (the two-finger algorithm computes new addresses while objects are being moved.)

Table-based Methods

These methods need no extra memory to keep live regions in order. Instead holes between live regions are used to temporarily store a table of information on where the regions have been moved to.

The table-based method of Haddon and Waite [HW67] starts by sliding live regions towards the top of the heap. The table is used to log the original starting address of each region and how far it is moved. To improve performance, the table is sorted according to the start address of the regions. In the last phase all references are updated by looking up how far each object has been moved.

Threaded Methods

These methods do not scan the heap to find which pointers to update as the algorithms above do. During garbage collection, a cell in each referenced object is replaced by a reference to the head of a list containing all fields (not objects) referring to the object. The last element of the list contains the value which was replaced by the reference to the list. Thus, it must be possible to distinguish a value from a pointer to distinguish a pointer from the end of the list. The data structure is only used during garbage collection. All pointers are restored when the collection cycle finishes. Fisher [Fis74] was first to find an effective way of building the lists and restoring the object graph. Since the object graph is distorted during reference updates, the mutator has to be locked during that phase. Thus, these algorithms are not appropriate in real-time systems.

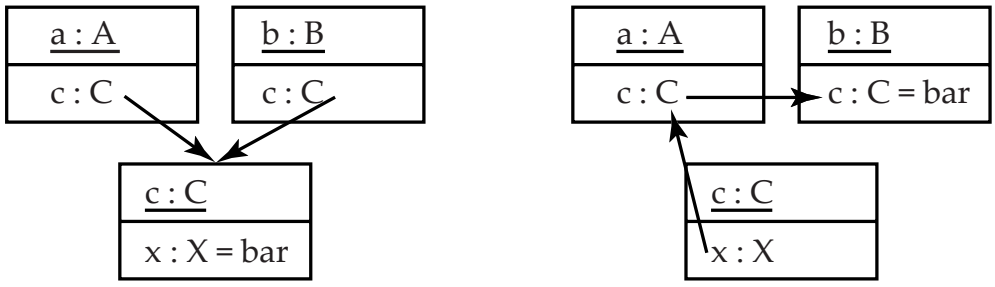
Figure 3.11 shows how objects are threaded. Objects A and B refer to object C. During reference updates, the field x in C is replaced by a pointer to the field in A which references C. The reference in A is then replaced by a pointer to B's reference to object C. The reference in B is finally replaced by the original value of x. This list is then traversed to update references to C.

3.4.2 Steele's Incremental Mark-and-Compact Algorithm

Steele has presented an incremental mark-and-compact technique that is designed for multi-processor systems consisting of one list-processing processor and one garbage collector processor [Ste75, Ste76].

The algorithm is a two-finger algorithm, but it also keeps free-lists and can be used in a non-compacting mode if the compaction and pointer update phases are disabled.

The mark phase is less conservative than Yuasa's. In the write-barrier, the object which contains the left-hand side reference is colored grey if the assignment would create a black to white reference (see Figure 3.12 and 3.12). Thus, a grey to white reference is created and the new object



(a) Mutator Execution (b) During threading

Figure 3.11. Threading objects

```
algorithm write-barrier(Object lhsParent, int offset, Object rhs)
    { Color rhs grey if it is white and the parent is black }
    lhsParent.data[offset] ← rhs
    if phase = marking then
        if lhsParent.isBlack() and rhs.isWhite() then
            lhsParent.mark ← false
            gcStack.push(lhs)
        end if
    end if
end
```

Figure 3.12. Steele's mark-and-compact write-barrier

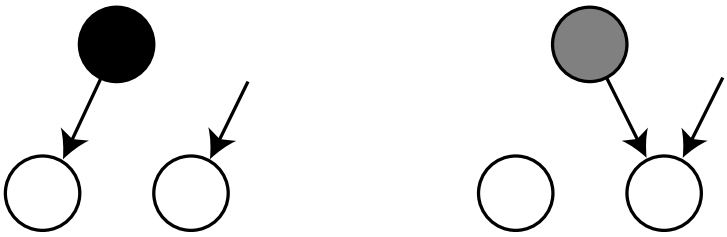
might be reclaimed if the reference is discarded before the end of the cycle. Thus, the GC takes a step back by changing a black object into a grey one, but this makes the algorithm less conservative.

Steele's write-barrier takes three arguments: a reference to the objects where the reference will be stored, the offset of the field where the reference will be stored, and the reference which will be stored. This can not be applied to local references. Local references are stored on the run-time stack. Thus, they will be marked when the roots are scanned.

3.4.3 Bengtsson's Mark-and-Compact Algorithm

Bengtsson describes an incremental mark-and-compact algorithm for use in real-time systems [Ben90]. The heap is divided into three areas. Area A should be large enough to hold all live objects. Areas B1 and B2 should be equally sized and large enough to keep all objects which are allocated during the period of a GC cycle.

To eliminate the reference update phase, all objects are accessed via an



(a) Before (b) After

Figure 3.13. Steele's write-barrier

object table. Thus, only references in the object table have to be updated. To find the reference in the table, each object keeps a pointer to its reference in an extra field.

Marking is done using a mark stack. Bengtsson uses the same write-barrier as Dijkstra does, i.e. the left-hand side is shaded before the assignment (see Figure 3.8 on page 27.) When the mark stack is empty, there are no more grey objects and the mark phase is finished.

The garbage collector now goes into its compaction phase. During compaction areas A and B1, or A and B2 are compacted into A. A and B1 are compacted during even GC cycles, and A and B2 is compacted during odd GC cycles. Compaction is performed by scanning the areas involved. When a marked object is found, it is moved into the lowest free address in A, and the reference in the object table is updated. If an unmarked object is found, it is ignored. While memory is compacted, allocation is done in the B area which is not being compacted. This region is empty.

As soon as memory is compacted, allocation can be done in A. When A is exhausted, a new cycle starts, and allocation is continued in B1 during odd GC cycles, and in B2 during even GC cycles.

3.5 Copying Algorithms

A copying garbage collector uses a heap which is divided into two or more sub-heaps. This section describes two sub-heap versions.

The two sub-heaps are labeled to-space and from-space, respectively. All objects are allocated in to-space where all live memory regions reside. When to-space is full, a flip is performed. First the labels are swapped, i.e. to-space becomes from-space and from-space becomes to-space. Next, the roots are copied from from-space (previously called to-space) into to-space. When an object is copied, all children of that object are copied too. When

all live objects have been copied, all pointers have to be updated to point to the new copies of the objects. Finally the garbage collector hands over control to the mutator.

An advantage of a copying garbage collector is that when the objects are copied, they are compacted. Thus, a copying garbage collector does not suffer from external fragmentation. Because the memory is compacted and placed at one end of the heap, allocation of n bytes can be done by simply sliding a pointer n positions in the free memory region.

An advantage of the technique is that the running time of the garbage collector is proportional to the number of live objects. Thus, a large heap size does not affect the running time of the collector, and the collector can be run less frequently. Appel [App87] has demonstrated that using a "standard" computer with a heap size seven times larger than the amount of live memory, a copying garbage collector is faster than stack allocation! However, Miller and Rozas [MR94] have demonstrated that stack allocation can be performed even faster than allocation using copying garbage collection on some platforms.

3.5.1 Cheney's Algorithm

A naive implementation of a copying collector would be recursive. Recursive implementations have the disadvantage that the depth of the runtime stack is crucial. An elegant iterative copying collector is described be Cheney [Che70].

A GC cycle starts by copying the roots into to-space. When an object is copied, the new address is stored in the from-space copy. Since the from-space copy will never be referred to again, the forward address can overwrite data in the object. Thus, no extra field is needed.

Objects in to-space are grey or black. To separate grey objects from black ones, a pointer named `scan` is used. Objects to the left of `scan` have been scanned and are thus black. The object that `scan` points to is being scanned. While an object is being scanned, its children are examined. If a child reference refers into from-space and the object has not been copied, it is copied into to-space and the child reference is updated to refer to the to-space copy. If the child has been copied, the child reference is updated to refer to the to-space copy using the forward address stored in from-space. When all roots have been copied and all copied objects have been scanned, the collector is finished.

The algorithm is a stop-and-copy algorithm, because it stops the mutator to perform garbage collection. Thus, it may interrupt the mutator for long periods of time which makes it unsuitable for use in most real-time systems.

```
algorithm read-barrier(Object in out obj)
    { Copy the object into to-space if it is in from-space }
    obj ← obj.copy()
end
```

Figure 3.14. Read-barrier in Baker's copying algorithm

3.5.2 Incremental Copying Algorithms

A major problem when interleaving garbage collection work with the mutator is synchronization. The mutator must not update an object which is being copied. Thus, the collector has to lock the mutator when copying an object, or be prepared to restart the copying if the object is updated. Unconditionally locking the mutator can be fatal in real-time systems, if object size is not limited. If copying is to be restarted, it is difficult to guarantee the collector's progress.

Baker's Copying Algorithm

This incremental copying algorithm is described by Baker [Bak78b]. In a non-incremental copying algorithm, objects in from-space are white, and the objects in to-space are grey or black depending on whether their children have been found or not. When using an incremental update technique the mutator must not create any black to white references (if there are no other references from a reachable non-black object to the white one.) To ensure this property, a read-barrier copies all objects seen by the mutator. Since no white objects can be seen by the mutator, there can be no black to white pointers. Copied objects are found using a forward pointer as in Cheney's algorithm.

Cheney's algorithm allocates new objects in the memory region directly after the objects which have been copied. While allocation is interleaved with copying in an incremental algorithm, it is impossible to tell at which memory position the allocations should start. Thus, Baker chose to allocate objects from the top instead.

Baker has chosen to allocate all objects black, thus in to-space. New objects are allocated from the top of to-space at the same time as objects are being copied to the bottom of to-space. A flip has to be performed when the two regions meet. It is crucial that all live objects are copied to to-space before the flip is made. If not, the application cannot continue.

Three pointers `scan`, `B` (bottom), and `T` (top), are used to divide to-space into grey, black, and newly allocated objects as shown in Figure 3.15.

Cheney's copy function has to be adjusted to move the objects to the B pointer (and increment it.) It also has to check whether `B` passes `T`, which should cause the system to be aborted. The allocation function adjusts the `scan` and `T` pointers.

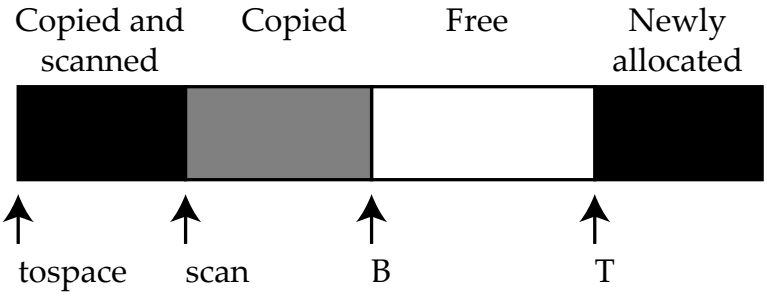


Figure 3.15. The layout of to-space in Baker's algorithm

The collector must also copy objects which are not accessed. Baker's algorithm does this when new objects are allocated. Each time an object is allocated, k grey objects are scanned, and their children are copied. The value of k has to be carefully chosen. If it is too small, the application will abort because a flip will be performed before all live objects are copied into to-space. On the other hand, if k is too large, the application will not run smoothly due to long interrupts.

Brooks' Algorithm

Many garbage collection algorithms have been optimized by using a write-barrier instead of a read-barrier. This improves the performance in most cases, because read operations are generally more frequent than write operations. Brooks [Bro84] modified Baker's algorithm to use a write-barrier and a very lightweight read-barrier, instead of an expensive read-barrier.

Baker chose to copy every object seen by the mutator to ensure the no-black-to-white-pointer property. Thus, the mutator can never see a white object and no black-white pointers can be created. Brooks found it too expensive to risk copying an object at every read operation. If no read-barrier is used to copy objects seen by the mutator, the mutator must be able to access objects in both to- and from-space. To ensure that no black-to-white pointers are created, the write-barrier copies left-hand side objects into to-space (in a way similar to Dijkstra and Bengtsson.) While all references refer to grey or black objects, no black-to-white pointers can exist.

The mutator refers to all objects via a forward reference stored in each object. If the object has been copied, the forward reference refers to the to-space copy, otherwise it refers to the from-space copy. The forward reference must be stored in a separate field, because it is impossible to distinguish a forward reference from any other reference. Thus, the penalty for the faster read-barrier is extra space for the forward references and an indirection at access (the read-barrier.)

```
algorithm write-barrier(Object out lhs, Object rhs)
    { Copy rhs to to-space if it is in from-space }
    if rhs.inFromSpace() then
        if not rhs.isEvacuated() then
            Object newCopy ← rhs.copy()
            B ← B + rhs.size
            rhs.forward ← newCopy
            newCopy.forward ← newCopy
        end if
        rhs ← rhs.forward
    end if
    lhs ← rhs
end

algorithm read-barrier(Object in out obj)
    obj ← obj.forward
end
```

Figure 3.16. Brooks' read and write barriers

3.6 Generation Scavenging

By studying lifetimes of objects, it has been shown that most objects die shortly after their allocation [Ung84]. Objects which have lived longer are less probable to die. The idea of generation based collection is to scan young objects more often than older ones. This can be done by extending the copying technique, as described above, to use more than two subheaps. Other techniques can also be used, as the generational GC based on mark-and-sweep in Section 9.1.1, but most generational collectors are based on copying collectors. This section will only describe garbage collectors based on the copying technique.

3.6.1 Inter-generational References

When a generation is collected, references from other generations have to be taken into account. Usually all younger generations are collected when an older one is collected. Thus, references from younger generations need no extra attention, but references from older generations to younger ones have to be considered. Several techniques to solve this problem have been proposed. These include:

Entry Tables

Each generation keep an entry table [LH83]. When a reference to an object in a younger generation is stored in an object of an older generation,

a write-barrier stores the reference to the younger object in its entry table. The reference stored in the older object is the reference to the cell in the entry table. A read-barrier is used to detect references to younger generations, and returns the value of the cell in the entry table. When a generation is collected it has to take references in the entry table into account.

Remembered Sets

It is expensive to use an extra indirection to access objects in younger generations. Using remembered sets [Ung84], objects in older generations which refer to objects in younger generations are remembered. Pointers to objects in older generations that refer to objects in a younger generation are stored in remembered sets of the younger generations. To prevent duplicate data in a remembered set, a bit in each object is used to indicate whether the object has been stored in a remembered set or not. When collecting garbage in a generation, objects in its remembered sets are also scanned.

Sequential Store Buffers

The sequential store buffers [HD90] technique was developed to simplify the write-barrier. Pointers to all potential pointers to younger generations are saved in a fixed sized buffer. The write-barrier can thus be very small and efficient. A protected page after the sequential store buffer is used to indicate a buffer overflow, which further simplifies the write-barrier. The garbage collector is activated when the buffer has overflowed. The garbage collector filters out uninteresting pointers, i.e. pointers which do not refer to references to objects in younger generations, from the buffer. Interesting pointers are saved in a hash table. Since a hash table is used, duplicate entries are prevented. The resulting hash table can now be used as a remembered set.

Card Marking

When using card marking [AKPY98], generations are split into equally sized cards. The size of a card is usually a power of two. Each time a reference is written to an object, a dirty bit in a bit vector is marked. The bit to mark is decided depending on which card is updated. The bit vector is called a modification bit table (MBT). Each generation has its own MBT. The write-barrier is slightly more complicated than that of a collector using sequential store buffers, but the need for a buffer is removed. At garbage collection time, the cards marked by a modification bit are scanned. The bit is cleared if the card does not contain a reference to an object in a younger generation, otherwise the bit is left marked for later collections.

3.6.2 Promotion Strategies

The age of an object is decided by which sub-heap the object resides in. A generation based collector has to apply a strategy to decide which objects should be promoted to an older generation and when to do it. A simple solution is to promote all live objects each time the sub-heap is collected. In addition to its simplicity, no age recording has to be done and only the oldest generation needs to be divided into to- and from-space. All other generations can be regarded as from-space when they are collected. The next older generation is considered as to-space. A disadvantage is that it is probable that most objects survive at least a couple of GC cycles. Thus, more generations are needed, otherwise most objects will end up in the oldest generation. Having many generations is not only a memory issue, it also means that more inter-generation references will be caught by the write-barrier, which is expensive.

If objects are to be kept in the same generation for several GC cycles, each generation needs a from- and a to-space. One possibility is to promote all objects within a generation. Using that scheme, there is no way to tell when a promoted object has arrived to the generation, i.e. relatively young objects can be promoted. To ensure that only older objects are promoted, age information can be kept in each object. This is costly, since it requires an extra field in each object. Another solution is to store age information in references to objects. Shawn [Sha88] describes a system where each sub-space, i.e. to- and from-space, is split into buckets. Buckets act like sub-generations. Every n:th GC cycle, objects are promoted to the next bucket. Objects in the last, i.e. oldest, bucket are promoted to the next generation. If m buckets are used in each generation, an object has to survive at least $n * (m - 1) + 1$ GC cycles to be promoted.

3.6.3 The Train Algorithm

The train algorithm [HM92, SG95], by Hudson and Moss, is a generational garbage collector where only part of the oldest generation is collected at every "full" invocation. Using the standard generational approach the entire heap needs to be collected occasionally, which causes longer interrupts. Using the train algorithm, the oldest generation is divided into trains, and the trains consist of cars. The algorithm guarantees that all cycles end up in a single train, which is reclaimed if no external references refer to it, and only one car needs to be collected at every "full" invocation.

3.6.4 Beltway Collectors

Beltway collectors [BJMM02], by Blackburn, Jones, McKinley, and Moss, generalize copying garbage collectors. They can be configured at runtime

to act like most[1] copying garbage collectors. Objects are grouped into increments that can be independently collected. The increments are organized into belts, and the increments of a belt are collected in first-in-first-out order. They present configurations of beltway collectors that acts like several existing copying garbage collectors and present two new collectors that outperform the competition.

3.7 Replication Copying

Synchronization is expensive in a copying garbage collector. In a replication copying system, the mutator only sees from-space objects, i.e. the opposite of Baker's algorithm where only to-space is visible to the mutator. Since only from-space is visible to the mutator no synchronization with the collector is required.

Objects which have been copied into to-space have to be updated when the from-space version is modified. A write-barrier can be used to modify the to-space object, or to keep an update log, which is used to update to-space objects before a flip is performed (see Figure 3.17.) When to- and from-space are flipped, the root references are updated to refer to objects in to-space. As in other copying algorithms, references between objects have to be updated. This can be solved by any technique which does not overwrite any data in from-space objects. An extra field can be used to store the forward address of a from-space object. If space is crucial, the forward pointer can be written over header data, but then this pointer has to be followed to access header information. Of course, the system must be able to distinguish a forward pointer from other header data. A bit indicating whether the object has been evacuated is one of the solutions which can be used.

Several variations of this technique have been proposed. These include Nettles et al. [NOPH92] and Nettles and O'Toole [NO93].

3.8 The Treadmill

A disadvantage of copying algorithms is that objects have to be moved. This is expensive, especially for applications which allocate memory aggressively. The treadmill algorithm due to Baker [Bak92] is a non-moving version of a Baker's copying algorithm [Bak78a].

To-space and from-space form two sets of objects in a copying garbage collector. These sets need not be implemented as two sub-heaps, they can

[1] Beltway collectors encompass all copying garbage collectors the authors of the publication are aware of, but they have left the train algorithm to possible future work

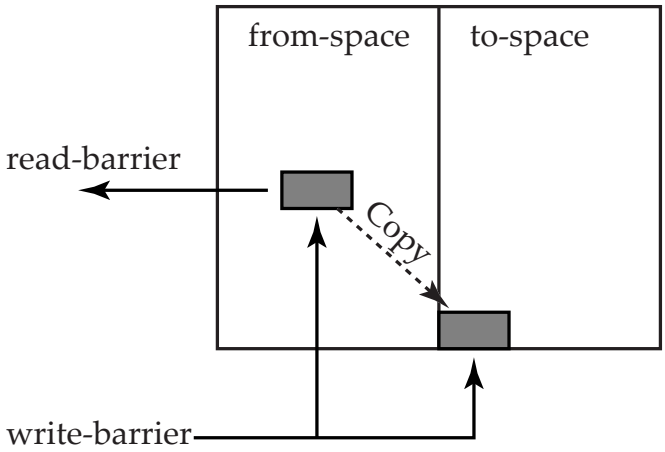


Figure 3.17. Replication copying

be implemented as two lists: one to-list and one from-list. Instead of copying the objects, the objects are moved from one list to the other. If the lists are doubly-linked, this is a constant time operation. Objects which are left in the from-list when all live objects have been moved are considered free, and can now be added to the free-list. Allocation is expensive in the original version of the treadmill, because the free-list must be searched for an object of appropriate size. However, if all objects are equally sized, allocation becomes a fast constant time operation. Techniques handling different sized objects better include Brent [Bre89], and Wilson and Johnstone [WJ93].

The treadmill concatenates the to-list, the from-list, and the free-list into a circular structure. Three pointers are used to divide the ring into from-list (`B`), to-list (`T`), and the free-list (`free`). A fourth pointer (`scan`) is used to mark the last black object of the to-list.

Since the treadmill has evolved from Baker's copying algorithm, the names of the pointers are analogous. The pointer `T` refers to the first object in the to-list, i.e. the top of the from-list. `B` refers the first object of the from-list, i.e. the bottom of the from-list. Objects from `B` to, but not including, `T` are colored ecru (off-white.) Grey objects reside between `T` and `scan`, including `T` but not `scan`. Objects between `scan` and `free`, not including the object pointed to by `free`, are black. Objects between `free` and `B`, not including `B` are white. The white objects form the free-list. This is illustrated in Figure 3.18.

The collector scans grey objects, and moves their white children either to the head or the tail of the list of grey objects. If white objects are moved to the tail, a depth-first traversal is performed, if they are moved to the head, a breadth-first traversal is performed. To distinguish white objects

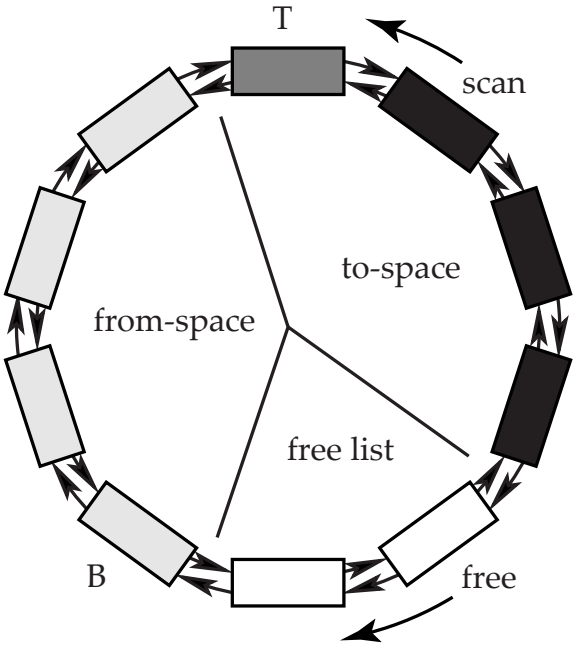


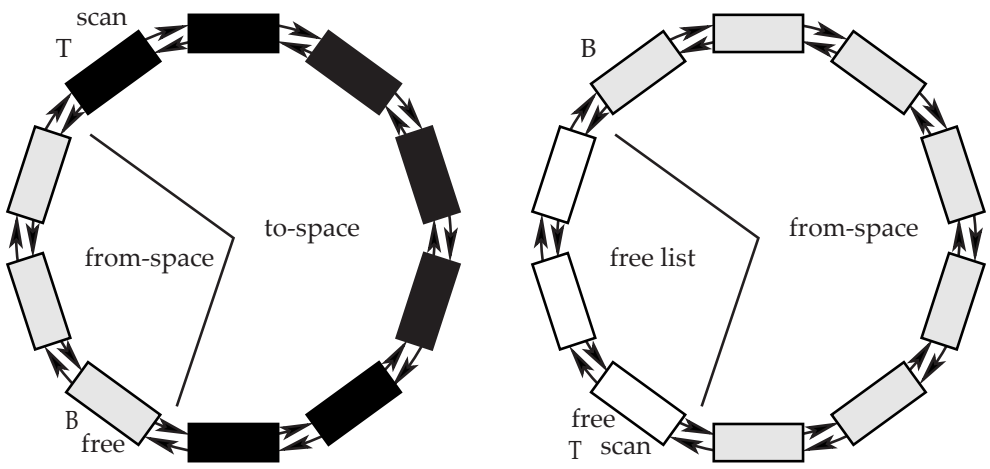
Figure 3.18. Structure of the treadmill

from non-white objects, a single mark bit is used. When a grey object has been scanned, the `scan` pointer is advanced towards `T`. When `scan` meets `T`, all live objects have been processed.

Allocation is done by advancing the `free` pointer towards `B`. If the free-list contains different sized objects, the list has been searched. The object which is allocated must then be inserted behind the head of the free-list. When `free` meets `B` the memory is exhausted. If all objects have been moved, a flip is performed. Otherwise, the collector has to scan all grey objects or insert additional objects into the treadmill if more memory is available.

`T` and `B` are swapped when the treadmill is flipped. The mark bit must also be reinterpreted so ecru means white and black means ecru. The roots are then moved to the grey region, i.e. between `scan` and `T`. This is done by removing the objects from the from-list and inserting them between `T` and the ecru objects. `T` is then set to point to the root object next to an ecru object, and `scan` is set to point to the grey object next to a white object, as described in Figure 3.19. With some imagination flipping can be seen as turning the circular structure. This is why the algorithm is called the treadmill.

The benefits lie in the properties inherited from the copying algorithm: only live objects are traversed and fast allocation (if only equal sized objects are used.) The main disadvantage is the space overhead. The space overhead is due to the fact that objects are stored in a doubly linked structure. The structure has to be doubly linked, otherwise advancement and movement can not be performed effectively.



(a) Before (b) After

Figure 3.19. Flipping a treadmill

3.9 Hardware-Supported Garbage Collection

GC algorithms implemented in hardware can be used to increase performance. It is important to keep in mind, that predictability is not achieved by this alone. Nilsen [NS93] has presented an implementation in hardware. Simulation indicates a worst case interruptions of 1 μs. Unfortunately, the implementation does not guarantee memory availability.

In any case, specially designed hardware has not been successful to date, particularly regarding portability, which is becoming increasingly important, also in the real-time domain.

3.10 Requirements of a Hard Real-Time GC

The chief requirement is that a hard read-time garbage collector is predictable. It must be predictable in both execution time and memory usage, i.e. given the requirements of a system it must be possible to calculate how much memory is required and how much work is needed by the garbage collector to keep the system from running out of memory.

As a secondary goal, the garbage collector should be fast and use a small amount of memory. Here it is the worst case that is most important, because that is what has to be used when designing the system. A good average performance is also nice, but not as important.

To be able to guarantee memory availability, external fragmentation must be taken into consideration. External fragmentation occurs when free memory is split up into fragments that can not be used by new objects even if enough free memory exists. The fragmentation must be predictable, otherwise it is too difficult to guarantee that the system does not run out of memory.

3.11 Algorithm Analysis

When a real-time garbage collector is to be developed, one can either start from scratch or use an existing technique as the basis for the new one. To our knowledge, all successful garbage collection techniques presented the last 10 years are improvements of the base GC techniques. You could even say that all successful garbage collectors are based on reference counting, mark-and-sweep, mark-and-compact, or copying garbage collection, which where all invented in the 60's. Thus, the odds of succeeding are probably better if an existing GC technique is used as the basis for a new one. The following sections discuss the whether the techniques can be improved to fulfill our requirements of a real-time garbage collector.

3.11.1 Reference Counting

Reference counting garbage collectors have the advantage of reclaiming objects as soon as they become unreachable, and does not work in cycles as the other collectors do. Thus, no memory is needed to allocate objects while the collector fulfills its cycle. Another advantage is that locking is extremely fine grained, i.e. they only lock the system for very short periods of time. Finally, the objects do not move around in memory, therefore no execution time overhead is caused by copying. This also results in fast access.

On the other side, reference counters can not reclaim dead cyclic data structures, since the objects in the cycle keep each other from decrementing their reference count to zero. Another disadvantage, caused by the advantage of not moving objects, is that memory is not compacted, thus external fragmentation may be a problem.

3.11.2 Mark-and-Sweep

Mark-and-sweep collectors also use very fine grained locking, thus interruptions from the GC are very short. Another advantage is that objects do not move around in memory.

However, since objects are not moved, mark-and-sweep collectors suffer from fragmentation as reference counting collectors do. Another prob-

lem is that the collectors work in cycles, thus objects are only reclaimed in the end of the cycle. It is common that some dead objects float from one GC cycle to the next, e.g. objects that are allocated in one cycle can seldom be reclaimed in the same cycle. Since objects are only reclaimed in the end of the cycle, the memory must hold all objects that were not collected during the last cycle plus all objects that are allocated during the current cycle. This does not only increase memory usage, but also make it harder to predict how much memory is required to run the system. To decrease the memory overhead, the garbage collector must finish cycles faster, i.e. do more GC work, and thus increase the execution time overhead. A good balance between execution time and memory usage is achieved when using a third of the heap to store objects allocated during the current cycle, i.e. an overhead of 50 %. When using a smaller part of the heap for new objects, the overhead of execution time increases rapidly. The execution time overhead is $O(1/memory\ overhead)$.

3.11.3 Mark-and-Compact

Mark-and-compact garbage collectors have the advantage of compacting the memory, which eliminates external fragmentation and improves the speed of allocation.

However, since memory is compacted, objects must be copied, which cause an execution time overhead, both while copying and while accessing objects. It may be possible to update all references to a copied object, but the cost of those updates can be even higher than the cost of accessing copied objects via handles. Copying objects may also cause the GC to lock the system for long periods of time. If a copy operation is allowed to be interrupted, it has to be restarted to guarantee that the latest version of the object is copied. A write-barrier, as used in replication copying GC, could be used to prevent this, but it is expensive. The GC work needed to complete a cycle is hard to calculate if copies are allowed to be restarted. Mark-and-compact also shares the problem of working in cycles, as all collectors except reference counters do. This results in large memory overhead.

3.11.4 Two Sub-Heap Copying

The main advantage of copying collectors is that the execution time of a cycle is only dependent on the size of the live objects, not on the size of the heap. However, this advantage is lost if finalization is to be supported. Finalization is a language feature that makes it possible to call a user defined cleanup method before the memory occupied by the object is reused. Most systems do well without finalization. Another advantage is that memory is compacted.

On the down side, objects need to be copied and the collector works in cycles, which cause a number of disadvantages (see above.) Another problem is that there is no way to continue if the system runs out of memory. Other techniques can often release some memory and then continue, but a copying collector gets stuck, since the GC cycle can not run to its end. Further, only half the heap can be used at any time, thus the memory requirement is doubled.

3.11.5 Generational Scavenging

Generational garbage collectors are often fast, since only parts of the heap is collected at most invocations. However, most generational collectors have to collect the entire heap sometimes. The exceptions either require complicated (and therefore slow) barriers to maintain inter-generational references, or may allow large amounts of garbage to float between GC cycles (train algorithm.) Thus, we have the choice of slow barriers, collecting the entire heap, or use methods that require more memory (that may be very hard to estimate.) The first two choices give a worse WCET than using the technique that is used to collect the generations, and the third give large and hard to predict memory usage. Since hard real-time systems have to calculate with the WCET and not the average execution, generational scavenging seems to be a poor choice.

3.11.6 Replication Copying

Replication copying garbage collectors have the advantages of copying garbage collectors plus that no synchronization is needed when accessing objects. Unfortunately, it also shares the disadvantages (except for the synchronization) and has a very complicated write-barrier. It may be possible to make a really effective solution using special purpose hardware. However, software solutions tend to be slow. The exception is when using functional languages where destructive operations are rare or prohibited, i.e. live data is seldom or never overwritten. However, destructive operations are common in the languages we target.

3.11.7 Baker's Treadmill

The treadmill shares many of the advantages of copying garbage collection without using separate sub-heaps, and thus without copying objects. Still, the treadmill does not suffer from fragmentation, since all objects are pre-allocated. Since objects are pre-allocated and of equal size, it may be difficult to use the treadmill in systems where the objects sizes vary, and may be unknown at compile-time. The solutions for this is commonly to

use multiple treadmills, but that decrease the possibility of reusing memory and depends on the developers to be able to predict the maximum number of objects of different sizes. Another disadvantage is the large per object overhead of keeping the objects in a double-linked list. An interesting alternative would be to construct larger objects by linking a number of smaller ones as files are constructed from blocks in a file system.

3.11.8 Hardware Solutions

Using custom has rarely been a success historically. It makes development dependent on special platforms, and the systems become difficult to port. These points plus the fact that we lack the knowledge to develop the hardware eliminates a hardware solution as an alternative.

3.12 Summary

When selecting a base technique, we started to remove candidates that we did not consider interesting. Two sub-heap copying collectors were eliminated since they use too much memory. Generational garbage collectors were eliminated since their WCET is longer than the techniques they are based on. Replication copying was eliminated since it seems too expensive to update the replicas if updates are too frequent. The treadmill was eliminated because its large per object overhead combined with its fixed object sizes. However as stated above, it would be interesting to see how a treadmill garbage collector where larger objects are constructed from smaller ones could compete. Finally a hardware solution is not interesting, since it would limit us to special hardware. At a later stage it may be interesting to investigate whether our collector should be implemented in hardware, but at this stage the algorithms are more important.

This leaves us with: reference counting, mark-and-sweep, and mark-and-compact. Mark-and-compact collectors suffer from the disadvantage of copying objects. This may are may not be a problem. Copying large objects are expensive in time, and can cause long interrupts or it may complicate the calculation of the GC work needed to complete a cycle. Since we do not want to limit the object size, mark-and-compact garbage collection is eliminated.

Reference counting has the advantage that objects can be reclaimed immediately when they become unreachable and that it does not work in cycles. These advantages should be compared to the disadvantage of not being able to collect cyclic data structures. We chose reference counting because a low memory usage is a high priority in many hard real-time systems.

— Those are my principles. If you don't like them I have others.

Groucho Marx

Chapter 4

Real-Time Reference Counting

This chapter introduces a new real-time garbage collection technique called real-time reference counting (RTRC) [Rit01, RF02]. The main advantage of the technique is that it increases memory usage efficiency by about 50 % compared to previously published real-time garbage collectors. The chapter is concluded with a thorough comparison with the competing techniques.

The standard reference counting algorithm was chosen as the basis of our real-time garbage collector because:

1. it reclaims memory immediately when it becomes garbage
2. its work is not organized in cycles
3. it locks the system during very short periods of time

However there are still a number of issues that must be handled to make reference counting attractive for hard real-time.

4.1 Drawbacks of Standard Reference Counting

Recursive freeing When the last reference to a data structure is deleted, all objects in that structure are reclaimed. If the data structure is large, in the worst case all objects on the heap, this behavior causes long interrupts in the execution of the system. Since recursive freeing can occur anywhere when reference counts are decremented, e.g. at assignments, the WCET becomes very pessimistic. To be able to use reference counting in hard real-time systems, recursive freeing must be eliminated.

External fragmentation External fragmentation occurs when small regions of free memory exist between the allocated objects. Even though there is enough memory to allocate a new object, there may be no contiguous region that is large enough. Thus, the allocation fails even if there is memory available. External fragmentation rarely causes problems, since clever allocation strategies keep it small. Even if the heap becomes more fragmented, most systems have enough memory to continue. But to predict the external fragmentation in advance is a difficult problem with no known solution. In a hard real-time system, the worst case memory usage must be known in advance, thus external fragmentation must be handled.

WCET of allocation Allocators usually have a WCET that is proportional to the heap size or to the logarithm of the heap size. The average execution time is normally much shorter due to pools of blocks of common sizes, but the average execution time is of little use in hard real-time. Thus, execution time should be improved for the technique to be competitive.

Inability to reclaim cyclic garbage Since the internal references of cyclic data structures keep all reference counts above zero, the objects that are part of cycles can not be reclaimed. Many systems can be implemented to have no dead cyclic data structures. However, for the technique to be useful when dead cycles can not be avoided, it must be possible to reclaim them.

4.2 Eliminating External Fragmentation

There are a few techniques that can be used to decrease external fragmentation. By using clever allocation techniques, external fragmentation can be decreased in general, but the outcome can not be predicted. If a fixed set of fixed size blocks are used as a pool of memory regions, all blocks can be statically allocated. This eliminates external fragmentation, but objects of different sizes can not reuse the same memory. Thus, the end result is commonly a larger memory foot print, and the analysis to find out what to statically allocate is tedious.

Fragmentation can also be eliminated by compacting memory, e.g. using copying [FY69] or mark-and-compact [Sau74] techniques. However, these methods are often considered too expensive and/or unpredictable. Moving objects also makes interfacing to other systems harder. In the Java platform [TGJS96], this is solved by the possibility to lock objects in fixed positions. This could cause serious fragmentation problems if the locks are kept for longer periods of time. And most of all it makes the system unpredictable. Other solutions include prohibiting heap allocation or lim-

iting allocation into arenas, i.e. memory areas that are freed as a whole, as in the real-time Java specifications [fJEG00, Con00].

In RTRC objects are divided into equally sized blocks, which is also used in many file systems and virtual memory systems. This eliminates external fragmentation completely, but introduces internal fragmentation which occurs when the last block of an object is not fully used. An important difference between internal and external fragmentation is that internal fragmentation is predictable, but external is not.

There are several ways to combine blocks into an object, e.g. linked lists and trees. For performance reasons most objects should fit into one or two blocks, so a linked schema works well. However, large objects such as arrays can be allocated using index blocks or trees. In the remainder of this thesis we assume that the blocks are kept in a linked list, but other techniques work as well. The requirements are that member access should be predictable, that taking a step in the iteration through the blocks of an object should take constant time, and that it must be possible to disconnect a block from the object in constant time.

At first, this technique may seem expensive in execution time, but it compares well to other alternatives. Real-time techniques that compact memory needs one dereference to find objects. To eliminate the dereference operation, all references to an object needs to be updated when an object is moved. Although this may be possible, it can be very expensive. Updating all pointers could be done, e.g. using a stop-the-world solution, but it causes too long interruptions for most non-trivial real-time systems. Thus, if memory is compacted a dereference operation is needed. One dereference operation is also what is needed to access members in the second block of an object using the blocked technique. Accessing a member in an object consisting of three blocks also needs one dereference on the average. Thus, the execution time of memory access using compaction and blocks are comparable when objects are relatively small. However, accessing data in larger objects is more expensive using blocks. On the other hand, compacting the heap also takes time.

4.2.1 Selecting the Block Size

Selecting the size of the blocks is crucial to limit the loss in both execution time and internal fragmentation. You would like to find a size that makes most objects fit in one or two blocks, but still does not cause the internal fragmentation to be too large.

A thorough investigation of allocation behavior of Java applications has been done by Dieckmann and Hölze [DH01]. The average size of objects (non-arrays) in the SPECjvm98 benchmarks [spe98] ranges from 12 to 23 bytes, which fits into one or two 32 byte blocks (including the overhead of the blocks and the object header.) The arrays are larger, but less com-

mon. Only in two cases (`compr` and `jess`) would more than 10 % of the objects require more than 2 blocks (including arrays). Many of the arrays are strings that can be allocated statically and thus contiguously.

Siebert investigates the impact of using different block sizes in the Jamaica JVM [Sie02] using the SPECjvm98 benchmarks. His results show that 32, 64, and 128 gives the best performance. The difference between these choices is small. Thus, a block size of 32 bytes seems like a good starting point, since it gives the least internal fragmentation. It is worth noting that arrays are allocated contiguously if possible in Jamaica, which often is possible in short running applications such as the SPECjvm98 benchmarks. It is likely that more arrays will be fragmented in longer running applications. This may suit some block sizes better than others.

The best block size is application dependent. To tune the application, the allocation function can be used to produce statistics about how many objects of different sizes are allocated. This information can be used when selecting the block size. One should also consider the behavior of the cache when selecting the block size.

4.3 Eliminating Recursive Freeing

Lazy reference counting (see Section 3.2.1) can be used to eliminate the recursive behavior of the decrement operation. However, it has a limitation in that only one object size is allowed. However, by constructing objects from equally sized blocks, this limitation is eliminated. Since the allocator constructs objects from blocks of dead objects, it is not necessary to search the to-be-free list to find an appropriate memory region. If the blocks of the first object in the list are not enough more blocks can be taken from the next object. Thus, recursive freeing is eliminated.

4.4 Improving WCET of Allocation

The WCET of allocation is linear in the size of the allocated object. Better WCET than this can not be achieved if objects are initialized. The execution time compares well to that of compacting memory managers, which are often considered to have superior allocation performance. An incremental compacting allocator needs to perform some compaction work during allocation, then the object is allocated by incrementing a pointer in the contiguous free memory region, and finally the memory is initialized. When using Weizenbaum's lazy technique to eliminate the recursive behavior of freeing with a heap divided into blocks, the allocation takes blocks from the to-be-free list and releases their child references, connects them, and initializes the new object.

4.5 Manually Reclaiming Cycles

A weakness in standard reference counting is that cyclic data structures can not be reclaimed. Several techniques have been developed to work around this problem, but none is designed for real-time. An important question to ask is whether or not cyclic data structures need to be collected. In systems where it can be guaranteed that no cycles become garbage, no cycles need to be collected. However, cycles must be collected in the general case.

The following section discusses manual techniques to reclaim cycles. Manual techniques are often more efficient than automatic ones, but they suffer from the problem of the human factor. Since humans are known to make errors, manual techniques should be used with caution.

4.5.1 Manually Breaking Cycles

A straightforward solution is to manually break all cycles before they become garbage. For this to work, the programmer needs to know when cycles become garbage and how to break them. In a general program that can be hard, but in a hard real-time system, the programmer needs full control of the application. Therefore breaking cycles manually should be possible. Knowing where, in the code, cycles should be broken is not the only problem. It is just as hard to find the cycles. However, tools can be developed to help out. During development and testing a *garbage detector* could be used to find cycles. A garbage detector could be, for example, a mark-and-sweep collector that runs periodically. When dead objects are found they must be part of a cycle. These objects can, for example, be identified by the expression that created them. Such a tool could be used to verify that no dead cycles remain allocated. Note that the garbage detector should only be used during testing.

4.5.2 Weak References

The difference between normal (strong) references and weak references is that weak references alone do not keep an object alive. An object requires at least one strong reference to stay alive. Weak references can be implemented in a reference counter by not counting them.

To collect cycles, developers need to guarantee that all cycles contain a weak reference and that normal references form a spanning tree of the object graph. For example, in a doubly-linked list all backward references could be weak, and if the nodes of a tree need references to their parents the upward reference could be weak. Most data structures can be constructed so that normal references do not form cycles. The difficult task is to locate the cycles, especially when legacy code is being reused.

Weak references also needs support from the programming language. Somehow it must be possible to create weak references, possibly by annotating normal references. In the case of Java, weak references are already in the language.

4.5.3 Balloon Types

Balloon types [Alm97] were introduced by Almeida. A *balloon type* is a data structure with only one entry point, i.e. external objects can not refer to the internal objects of a balloon object. However, it is possible to refer to an internal object of a balloon type from a local variable or an actual parameter. A balloon type that does not allow any external references to its internal objects is called *opaque*. Almeida defines the concept with an invariant. If B is an object of a balloon type then:

1. There is at most one reference to B in the set of all objects.
2. This reference (if it exists) is from an object external to B.
3. No object internal to B is referenced by any object external to B.

Almeida also presents a static analysis to check the invariant.

A cycle within a balloon object can be collected by breaking it when the reference to the entry point is lost. Balloon types have a lot in common with Bobrows's groups (see Section 3.2.3.) In both cases objects are split into groups, and the inter-group references are counted separately. However, balloon types are easier to implement correctly.

In Java, an opaque balloon type can be implemented as a package with only one public class (the entry point). An object of this class must identify a unique data structure of the balloon type (a balloon object.) Unfortunately (in this case), references to objects of non-public classes can be returned outside its package. Developers must verify that this does not cause any problems. A finalizer in the entry point class can be used to break all internal cycles. This works since the entry point can not be part of a cycle (it can only be referenced by one other object). To summarize, in Java an opaque balloon type can be implemented as a package where:

- only one class is public
- objects of the public class uniquely identify a balloon object
- no references to internal objects are returned from the package
- the public objects are not part of any cycles
- a finalizer in the public class is used to break the cycle(s)

Note that internal cycles are still not collected. Considering this, cycles with different life span should be kept in different balloon types. Of course it must also be verified that separate balloon objects do refer to the internals of other balloon objects.

Balloon types are easier to work with than Bobrow's group, and they do not require any support from the runtime system. However, both techniques share the problem with cycles that are not properly nested.

4.6 Automatically Reclaiming Cycles

The techniques presented in Section 4.5 were all manual, and thus suffer from the problem of the human factor. However, when developing a hard real-time system the developers need to have full control of the application, so most such systems should be able to use manual techniques. If manual techniques are considered unsuitable for an application, the application may need to be redesigned. If the application still is to complex an automatic technique may solve the problem.

Automatic techniques suffer from the drawback that all memory is not reclaimed as soon as it becomes unreachable. This need not cause problems if the overhead in execution time and memory usage is predictable and low.

4.6.1 Mark-and-Sweep Backup

The technique presented in Section 3.2.2 does partial scans of the object graph to find dead cycles. It can not be used for hard real-time systems, since the sizes of scanned sub-graphs are unknown. This can be solved by using a full backup garbage collector. This garbage collector should share properties with the reference counter, so that the integration is as smooth as possible.

The real-time mark-and-sweep technique in Jamaica is a good example. The collector is non-moving, and the object layout uses the same blocked object layout as RTRC does. The attributes needed by the mark-and-sweep collector can often share memory with the reference counter, by reserving some high-order bits to the mark-and-sweep collector. Thus, no extra object overhead is required to merge the techniques. The execution time overhead is of course dependent on the implementation, but if the reference counter and mark-and-sweep bits are stored in the same word there will be large cache benefits, so it should be possible to keep the overhead low.

The main advantage of reference counting is that the memory overhead can be kept significantly lower than by using other techniques such as mark-and-sweep. One may think that this advantage is lost when a

constant BLOCK_COUNT ← 4096
constant BLOCK_SIZE ← 32

Figure 4.1. Configuration

backup mark-and-sweep garbage collector is used. This is partially true. A larger memory overhead is needed, but in most cases it can be significantly smaller than if mark-and-sweep was used alone. The reason for this is that objects that are not a part of cyclic data structures when they die will be reclaimed immediately by the reference counter, and will not waste space until the mark-and-sweep cycle is finished.

The amount of memory overhead that is required is the amount of memory that is occupied by dead objects that are reachable from cycles that may die during one mark-and-sweep cycle plus the objects that float between cycles. Thus, if 10 % of the objects that die during a mark-and-sweep cycle are reachable from dead cycles or float between cycles, then 10 % of the memory overhead used by a plain mark-and-sweep collector is required. To minimize the overhead further, all manual techniques presented above can be used to decrease the number of dead cycles in the system.

4.7 Design

In this section, the design of a RTRC is presented. This design does not cover recovering cyclic data structures, since any of the techniques described above can be used. As stated above, many systems can quite easily be designed not to produce any cyclic garbage, especially hard real-time systems where the developers must have full control of the execution of the system.

4.7.1 Configuration

Two constants are used to configure the garbage collector. The constant `BLOCK_COUNT` holds the number of blocks, and `BLOCK_SIZE` holds the size of the blocks. In Figure 4.1, the size of the blocks is set to 32 and the number of blocks is set to 4096. These are compile-time constants that can be tuned for the application. The block size can be set to any size larger than the object header (see below.) The size of the heap, i.e. the number of blocks, can easily be turned into a runtime constant, but changing the block size during runtime would require changes in the member access code.

```
record type
    integer size
    integer block_count
    procedure decchildren(type, object, int)
end

record object
    object next    { Next block }
    type type
    union  nr
        object next    { Next object }
        integer rc
    end
end

union block
    record object head
    byte[BLOCK_SIZE] data
end
```

Figure 4.2. Type declarations

4.7.2 Type Definitions

The types needed by RTRC are presented in Figure 4.2. The type `type` defines a record containing type information. The only information needed by the reference counter is the size of the objects of the type and a function that decrements the child references of a specific block of an object (`decchildren()`). The `decchildren()` method can be replaced by a data structure that holds information on where references can be found in the object. The size has to be given in both bytes (`size`) and blocks (`block_count`).

The `object` type defines the object header. The first field of the header is a pointer to the next block of the object (`next`). This field is also used to connect the blocks in the free list. The `next` field is a pointer to the type of the object (`type`). Next is a union (`nr`) containing the reference counter (`rc`) or the next field (`next`), which is used when the object is stored in the to-be-free list. The reference count is always zero when the object is in the to-be-free list, so it is not needed then. Note the difference between the `next` fields. The first is used to connect blocks in live and dead objects, and the other is used to connect objects in the to-be-free list.

Finally, the type of a block is defined. It is defined as a union of the type of the head of an object and a byte array type with `BLOCK_SIZE` cells. The full object header is only used in the first block of objects. In all other blocks, only the first `next` field is used.

The per object memory overhead on a 32-bit architecture is 8 bytes plus 4 bytes per block including the type information required by all modern object-oriented languages.

The real-time copying collector (see Section 9.1.2) uses one word to store type information and a forwarding pointer to the current copy of the object. Since all objects are contiguous, no block pointers are needed. However, to implement the `Object.hashCode()`, which is common to many modern languages such as Java and C#, method efficiently on systems using moving garbage collectors, an extra word per object is required. There are optimizations that only require one word if the method has been called and the object has been moved. This problem occurs since implementations of the `Object.hashCode()` method simply return the address of the object when using non-moving collectors. However, the address is not constant when using a moving collector, so the hash code has to be stored if the object is moved. Thus, the overhead is 12 bytes per object, i.e. the same as for RTRC for small objects.

The object overhead in RTRC is the same as in the presented real-time mark-and-sweep collector (see Section 9.1.3.) It keeps the type information in one word and the mark-bits in another. A mark-and-compact collector would only need the type information and a forwarding pointer as the copying collector does. However, since it moves the objects, the problem with `Object.hashCode()` must be solved.

4.7.3 Global State

The global state of the implementation is kept in the variables presented in Figure 4.3. These variables are internal to the garbage collector, so users never access them directly. The `freelist` keeps all blocks that are ready to be used by the allocator. These blocks should no longer refer to live objects. Most systems would gain speed if all non-header cells where initialized to zero before the blocks are added to the freelist.

The `available` variable stores the number of blocks in the free list. The `tbflist` keeps the to-be-free list. Here, all dead objects are listed before they are either moved to the free list or directly allocated to an object. Before blocks are reused, their references to other objects need to be decremented. When a block from the to-be-free list is needed, the first object of the list is removed and stored in the `head` variable. The `head` variable is then used as a list of blocks that are available to the allocator. The references stored in blocks are decremented when the blocks are removed from the head list. To be able to find the references, the type of the current object is stored in the `type` variable, and the block sequence number is stored in `blockseq`. Finally, an array called `heap` keeps all blocks available to the system.

```
block[BLOCK_COUNT] heap

integer available ← BLOCK_COUNT
object freelist ← heap
object tbflist ← null

object head ← null
type type ← null
integer blockseq ← 0
```

Figure 4.3. Global data

```
algorithm initialize()
    for i in 1..BLOCK_COUNT-1 do
        heap[i-1].head.next ← &heap[i]
    end loop

    heap[BLOCK_COUNT-1].head.next ← null
end
```

Figure 4.4. Initialization

Access to the global state is protected by mutexes in the code presented below. The synchronization is designed to be as fine grained as possible. Several synchronization blocks can be merged to make it coarser. In some cases synchronization can be avoided using atomic assembler instructions.

4.7.4 Initialization

The RTRC initialization connects all blocks into one huge free list. Depending on the platform it may or may not be necessary to initialize the heap with zeros first. Since the heap can be statically allocated, this is often done automatically. The initialization function is shown in Figure 4.4. Its WCET is linear to the number of blocks on the heap. This could be very expensive when using virtual memory, but the systems we are targeting do not use virtual memory, thus initialization is fast.

4.7.5 Allocation

Allocating blocks from the free list is done by traversing the free list until the requested number of blocks has been found, terminating the list of blocks, and adjusting `available` and `freelist`. The algorithm is presented in Figure 4.5. Its WCET is proportional to the number of blocks being allocated.

Allocating from the to-be-free list is slightly more complicated. The function is shown in Figure 4.6, and important data structures are shown

```
algorithm freelist_alloc(int size, Block out lastBlock) returns Object
    Block blocks
    Block tail
    synchronized available
        available ← available - size
    end synchronized

    synchronized freelist
        blocks ← freelist
        tail ← blocks
        freelist ← freelist.next
    end synchronized

    for i in 1..size-1 do
        synchronized freelist
            tail.next ← freelist
            tail ← freelist
            freelist ← freelist.next
        end synchronized
    end loop

    synchronized freelist
        tail.next ← null
        lastBlock ← tail
    end synchronized

    return blocks
end
```

Figure 4.5. Allocating from the free list

in Figure 4.7. The `head` variable keeps the object that is currently used to extract blocks from. If `head` is null, one object is taken from the to-be-free list. The type information is stored in `type`, `blockseq` is set to zero, and `blocks` is set to point to the first block of the object that is allocated. For each block being allocated, its child references are decremented using the `decchildren()` method in the type information. If we run out of blocks in the current object, the next object in the to-be-free list is taken and its blocks are used. The WCET of the function is proportional to the number of blocks being allocated, thus proportional to the object size.

The user function used to allocate objects allocates as many blocks from the free list as possible. The rest of the blocks are allocated from the to-be-free-list. Finally the lists of blocks are concatenated and the object is initiated. The function is presented in Figure 4.8. Its WCET is linear in the size of the object being allocated.

```
algorithm tbf_alloc(int size, Block out lastBlock) returns Object
    Block blocks ← head { The first block of the new object }
    lastBlock ← null    { The lastBlock of the blocks in the new object }

    for n in 1 .. size do
        synchronized tbf_alloc
            if head = null then
                if tbflist = null then
                    error("Out of memory")
                end if

                synchronized tbflist
                    head ← tbflist

                    if blocks = null then
                        blocks ← head
                    end if

                    type ← head.type
                    tbflist ← tbflist.nr.next
                    blockseq ← 0
                end synchronized
            end if

            if lastBlock ≠ null then
                lastBlock.next ← head
            end if

            type.decchildren(head, blockseq)
            lastBlock ← head
            head ← head.next
            blockseq ← blockseq + 1
        end synchronized
    end loop

    return blocks
end
```

Figure 4.6. Allocating blocks from the to-be-free-list

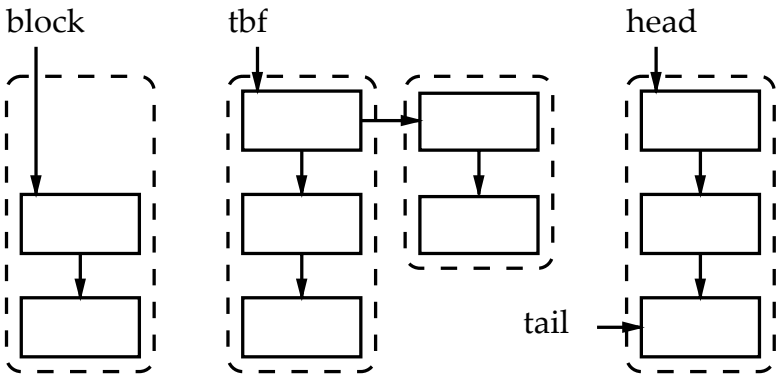


Figure 4.7. Data structures used by `tbf_alloc`. The dashed boxes represent objects and the solid boxes represent blocks. To the left is the free object from which the blocks are currently taken. In the middle are the objects in the to-be-free list and to the right is the object being allocated.

```
algorithm alloc(Type type) returns Object
    int size ← type.size
    Block tbfblocks ← null
    int fromfree ← min(size, available)

    if size - fromfree > 0 then
        tbfblocks ← tbf_alloc(size - fromfree)
    end if

    if fromfree > 0 then
        blocks ← freelist_alloc(fromfree, lastBlock)
        lastBlock.next ← tbfblocks
    else
        blocks ← tbfblocks
    end if

    blocks.type ← type;

    return blocks;
end
```

Figure 4.8. Allocating objects

```
algorithm prealloc(int size)
    if available ≥ size then
        return
    end if

    Block blocks ← tbf_alloc(size - available, lastBlock);

    synchronized freelist
        available ← size
        lastBlock.next ← freelist
        freelist ← blocks
    end synchronized
end
```

Figure 4.9. Pre-allocating blocks

4.7.6 Increasing Allocation Performance

It is sometimes beneficial to increase the execution speed of high-priority processes. In RTRC this can be accomplished by guaranteeing that all allocations of high-priority processes can be done from the free list. Thus, no time is spent decrementing child references. The runtime system provides a function (`prealloc()`), as presented in Figure 4.9, which allocates blocks from the to-be-free list and puts them directly into the free list.

When reserving pre-allocated blocks for high-priority tasks, the allocation function must always leave a specified number of blocks in the free list when allocating blocks to low-priority tasks. And after the execution of high-priority code, the `prealloc()` function must again be called to fill the free list with blocks. The number of blocks needed in the free list can be calculated using the technique presented in Henriksson's thesis [Hen98]. The worst case execution time of `prealloc()` is proportional to the number of blocks being pre-decremented.

4.7.7 Releasing References

The function that decrements reference counts (`release()`) is presented in Figure 4.10. The function decrements the reference count of the object, and adds it to the to-be-free list if the reference count falls to zero. The WCET of `release()` is constant.

4.7.8 Public Interface

The user interface to RTRC is simple. Users call `alloc()` to allocate objects and `prealloc()` to guarantee that a certain number of blocks are available in the free-list. The only other operations that are required are the operations the increment and decrement the reference counts. None of

```
algorithm release(Object obj)
    obj.rc ← obj.rc - 1
    if obj.rc = 0 then
        synchronized tbflist
            object.nr.next ← tbflist
            tbflist ← object
        end synchronized
    end if
end
```

Figure 4.10. Releasing objects

these functions are normally called directly by the users, since the compiler generates the necessary calls.

The information stored in the global variables is of little or no interest to the user. What could be interesting is the heap size and number of available blocks. However, it is important to note that the `available` variable does not hold the number of free blocks in the system, but the number of blocks in the free-list. There is no way to see how many blocks are in use. To get this information, a function similar to `prealloc()` but instead reclaims as many blocks as possible must be implemented. After this function has been called, the `available` variable holds the number of free blocks in the system. However, the execution time of this function is proportional to the heap size. A thread on the lowest priority level can execute this function if this kind of information is vital.

4.8 Complexity and Overhead

4.8.1 Execution Time

Initiating the memory manager is a matter of putting all blocks into a linked list. The WCET is linear in respect of the size of the heap.

Allocation (from the to-be-free-list or free-list) takes blocks from the respective list. When blocks are taken from the to-be-free-list their references must also be released. Since releasing references of a block takes constant time (the size of blocks is constant and thus the maximum number of references), the WCET of an allocation is linear with respect to the size of the allocated object.

Sometimes it is desired that allocation in high-priority tasks should be as fast as possible. This can be accomplished by pre-decrementing references of blocks in the to-be-free list and moving these to the free list. This is done by the `prealloc()` function. The worst-case execution time of this function is proportional to the number of blocks that should be pre-decremented.

Finally, both the increment and decrement operations have a constant worst-case execution time.

If a backup garbage collector is used, it needs to execute its read and/or write barriers as RTRC executes increments and decrements. With proper data layout, the cache will make the execution time overhead small. The backup collector would also need to run during allocation. The execution time per allocated block for this GC work is DC/o, where DC is the amount of memory that is occupied by dead objects reachable from dead cycles during one GC cycle and o is the amount of memory in blocks that is not used to store live objects (see below.)

4.8.2 Memory

The memory overhead can be divided into internal fragmentation, external fragmentation, the object header, and the extra memory temporarily needed to store new objects during a GC cycle.

The internal fragmentation is the extra memory that is unused "inside" the memory region allocated to an object. It is on the average half the size of a block per object, if all sizes of objects are equally probable. However, it should be possible to tune the block size to minimize this overhead in most cases.

The external fragmentation is the extra memory that can not be used between the memory regions occupied by objects. It is zero, since all blocks are of the same size and all blocks can be used to store objects.

The object header consists of type information, the reference counter (that can also be used to store mark bits), and the pointers that connect blocks into objects. On a 32-bit architecture, the overhead for one object is 8 bytes plus 4 bytes per block. The pointer to type information can be exchanged for an index into an array and the reference counter will in most cases fit into 16 bits. In smaller systems also the pointers that connect blocks can be 16-bit indexes. Using this schema the per object overhead sums up to 4 bytes plus 2 bytes per block.

No extra memory used to store objects during the GC-cycle is required if the system has no dead cycles, but if they do the amount of memory is selected to make allocation fast enough (as described above.)

Figure 4.11 illustrates an example of the memory layout of an application that uses RTRC, and Figure 4.12 present a diagram of theoretical estimate of how much memory is occupied by the different kinds of memory overhead in a system with object sizes of 11 – 33 bytes, which are common in Java systems [DH01].) The results show that 45 % of the memory can be used to store data (i.e. fields of objects), the rest is occupied by the runtime system. It may seem as a poor result, but it should be compared to 30 % for the real-time mark-and-sweep collector and 21 % for the real-time copying collector (see Figure 4.18 and 4.19.) A thorough comparison

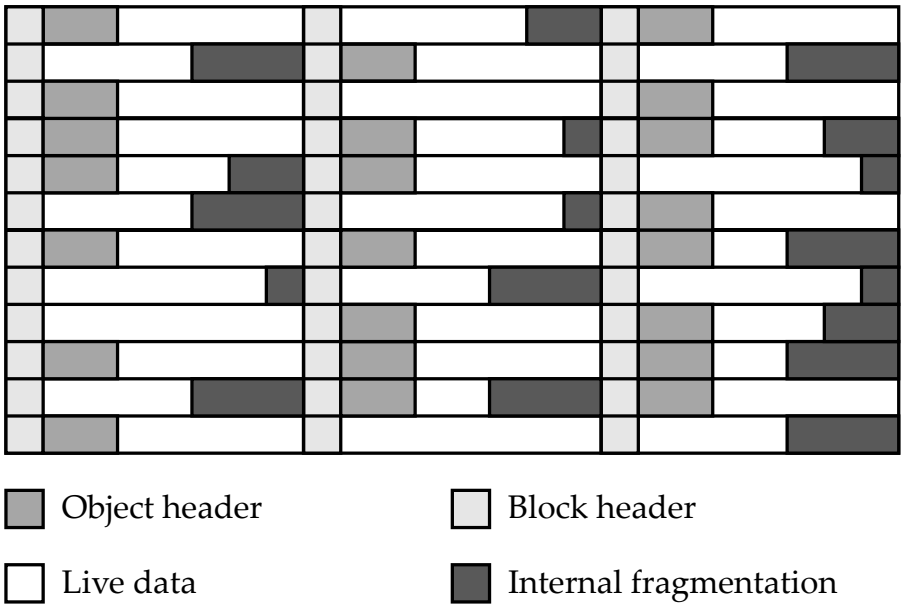


Figure 4.11. Memory layout using RTRC

is presented in Section 4.10. Internal fragmentation is the main contributor to the overhead of RTRC, which emphasizes the importance of selecting the block size.

4.9 Emitted GC Code

All incremental garbage collectors need to insert code to guard modifications to the object graph, so does RTRC. The fragments of code emitted by incremental garbage collectors are called barriers. Two kinds of barriers are used: read- and write-barriers (see Section 3.1.) A read-barrier is inserted where a reference is read, and a write-barrier is inserted where a reference is written. Since a reference counter is only concerned with operations that changes reference counts, read operations can be ignored. However, reference counts need to be updated when references are written.

4.9.1 Allocations

The allocation is described in detail in Section 4.7.5 and 4.7.6. It is normally best to initiate the reference counter to one, since a reference to the new object is normally stored in a variable. However, that is not always the case. Extra care is needed if no reference to a recently allocated object is stored. Consider the expression:

```
new TickerThread().start()
```

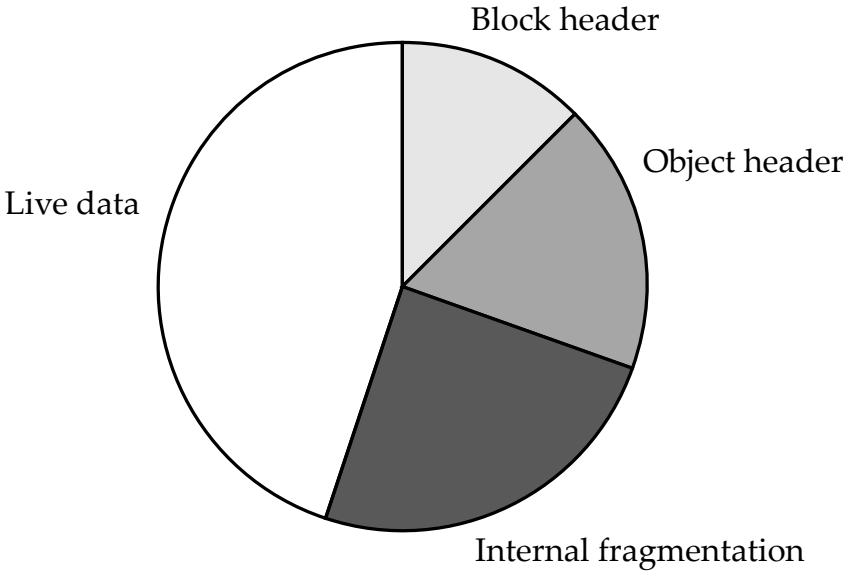


Figure 4.12. Memory usage using RTRC

The problem that arises is caused by the lack of a decrement operation that otherwise would be invoked when the variable is overwritten or goes out of scope. It does not help to initiate the reference count to zero, since still no check would be emitted to catch objects with a reference count of zero.

It is neither safe to emit code to reclaim the object after the expression has been evaluated, since a reference to the object could be stored while the expression is evaluated. A solution is to initiate the reference count to zero, and to check if the reference count is still zero after the evaluation. A simpler solution is to store the allocated object in a temporary variable. The disadvantage of that is that the object may be kept alive longer than necessary. A work around for this disadvantage is to set the temporary to null after the evaluation.

4.9.2 Reference Assignments

The objects that may be involved in an assignment are: the object referred to by the left hand side (LHS), and object referred to by the right hand side (RHS). Any of these references can, of course, be null. If the LHS objects exist, its reference counter should be decremented and the reference counter of the RHS object should be incremented if it exists. The increment operation should be invoked first, otherwise problem arise if both references refers to the same object and there are no other references to the object as shown in the following expression.

$$\text{Object a} \leftarrow \text{new Object()}$$
$$\text{a} \leftarrow \text{a}$$

```
algorithm write-barrier(Object in out lhs, Object rhs)
    if rhs ≠ null then
        rhs.rc ← rhs.rc + 1
    end if
    if lhs ≠ null then
        release(lhs)
    end if
    lhs ← rhs
end
```

Figure 4.13. Reference Assignment

Since a reference update of an assignment where LHS and RHS refers to the same object is redundant, it can be removed. However, this check should be performed at compile-time, since such assignments are extremely uncommon in real applications. The pseudo code of a reference assignment is presented in Figure 4.13.

It is also worth noting that `RHS` can be evaluated twice if care is not taken. It is probably easiest to store `RHS` in a temporary variable that is later used in the increment operation and assignment.

4.9.3 Method Calls

The objects involved in a method call are the objects referred to by the actual arguments and by the returned reference. The reference counters of the objects that are passed to a method should either be incremented just before the call or immediately when the method is entered. Since most (non-library) methods are called more than once, it is safe to assume that the code size decreases if the update is performed in the methods instead of at the call.

When a method that returns a reference is invoked, some work is required to handle the return value. First, if the return value is not stored in a variable, it has to be decremented explicitly (analogous to the problem with allocations).

The variable that will store the return value needs extra care. The object that it refers to before the call needs to be decremented. However, this can not be done before the method is invoked, because that could cause the object to be reclaimed prematurely. Consider the following code fragment:

```
Object a ← new Object()
a ← foo(a)
```

If `a` is decremented before the invocation, it has not been incremented as an actual parameter yet. Thus, it will be reclaimed. Therefore the old reference needs to be backed up and decremented after the method returns. The

```
algorithm invoke(Object in out result, Method method, Arguments args, ...)
    Object tmp ← result
    result ← method(args, ...)
    if tmp ≠ null then
        release(tmp)
    end if
end
```

Figure 4.14. Invoking a method that returns a reference to an object

pseudo code for a method invocation of a method that returns a reference to an object is presented in Figure 4.14. Invocations of methods that do not return a reference to an object need no extra code.

To summarize, since the arguments are handled inside methods, they can be ignored at the call. Only the variable that stores the return value needs to be taken care of and only if it is a reference.

4.9.4 Methods

To keep the objects referred to by the actual arguments of a method from being reclaimed, their reference counters are incremented as soon as the method is invoked. When the execution of a method is finished, e.g. due to a return statement or an exception, the reference count of objects referred to by local variables and actual arguments must be decremented. This may decrease the performance of exception handling. However, if exceptions are used for exceptional behavior, it will only cause a minor impact on the overall performance of the system. Figure 4.15 shows how a method, using RTRC, can be implemented in C. The code that needs to be emitted is the increments in the prologue, the cleanup code, and the modification of the return statements.

4.10 RTRC vs. The Competition

As stated above, reference counting has some disadvantages compared to other garbage collection techniques, and there do exist real-time copying and mark-and-sweep collectors. Therefore, the need for a real-time reference counting technique might be questioned. The following sections will explain the advantages of RTRC compared to the competition.

4.10.1 Execution time

Table 4.1 compares the worst-case execution time of the operations of the real-time reference counter presented in this paper to the worst-case execution times of other real-time garbage collectors [Hen98, Sie02]. Neither the

```
object_t *method(object_t *this,
                 object_t *arg1,
                 int arg2,
                 object_t *arg3)
{
    object_t *result = null;
    object_t *local = null;

    /* Increment the parameters */
    RC_INCR(this);
    RC_INCR(arg1);
    RC_INCR(arg3);

    if (arg2 > 0) {
        /* return arg1 */
        result = arg1;
        goto cleanup;
    }
    else {
        /* return arg3 */
        result = arg3;
        goto cleanup;
    }

cleanup:
    /* Decrementing locals and parameters except */
    /*  the value that is returned               */
    RC_DECR(this);
    RC_DECR(arg1);
    RC_DECR(arg3);
    RC_DECR(local);

    return result;
}
```

Figure 4.15. A method using RTRC implemented in C

Operation	RT-Copying	RT-Mark-and-Sweep	RTRC
Increment (or equal)	$O(1)$	$O(1)$	$O(1)$
Decrement (or equal)	$O(1)$	$O(1)$	$O(1)$
Allocation/Free	$O(s + \frac{1}{o})$	$O(s + \frac{1}{o})$	$O(s)$
Member access	$O(1)$	$O(s)$	$O(s)$
Array access	$O(1)$	$O(\log s)$	$O(\log s)$

Table 4.1. Worst case execution time of operations in real-time garbage collectors. The size of an object is denoted by s and the memory overhead by o.

copying nor the mark-and-sweep algorithm use increment or decrement operations. However, both use read/write barriers that perform equivalent operations. In the table, the worst-case execution time of the barriers are compared to the worst-case execution time of the increment/decrement operations. The major improvement of RTRC is the WCET of the allocation/free pair.

The execution time overhead is caused by the copying and mark-and-sweep collectors' inability to reclaim dead objects immediately. These collectors work in cycles, and memory is only reclaimed at the very end of the cycle. Thus, the system memory must hold the live objects, the objects allocated during the current cycle, and the dead objects that float between cycles. To keep the overhead low, the cycles can execute faster, but that in turn increases the WCET of allocation (since more GC work is needed to complete the cycle faster.) Since RTRC reclaims dead objects immediately, there is no such cost.

The execution time of RTRC is not dependent on the memory usage of the system, therefore all memory can be used without affecting the WCET. This is not possible using the other techniques due to the behavior of the allocation/free operations. Using all memory is especially advantageous for embedded systems.

Copying garbage collectors compacts the memory during garbage collection, which gives fast allocation and eliminates external fragmentation. However, since objects move around, a handle is used to access the object. This causes an extra pointer dereference which could be compared to accessing data in the second block using RTRC. Thus, if most objects can be stored in one or two blocks, accessing members is as fast or faster in RTRC compared to using a copying garbage collector.

Large objects, such as arrays, cause problems to all these techniques. Using a copying collector with large objects cause long interrupts while the object is being copied. To improve the response time, the system could interrupt a copy in progress, but that in turn causes problems in proving and

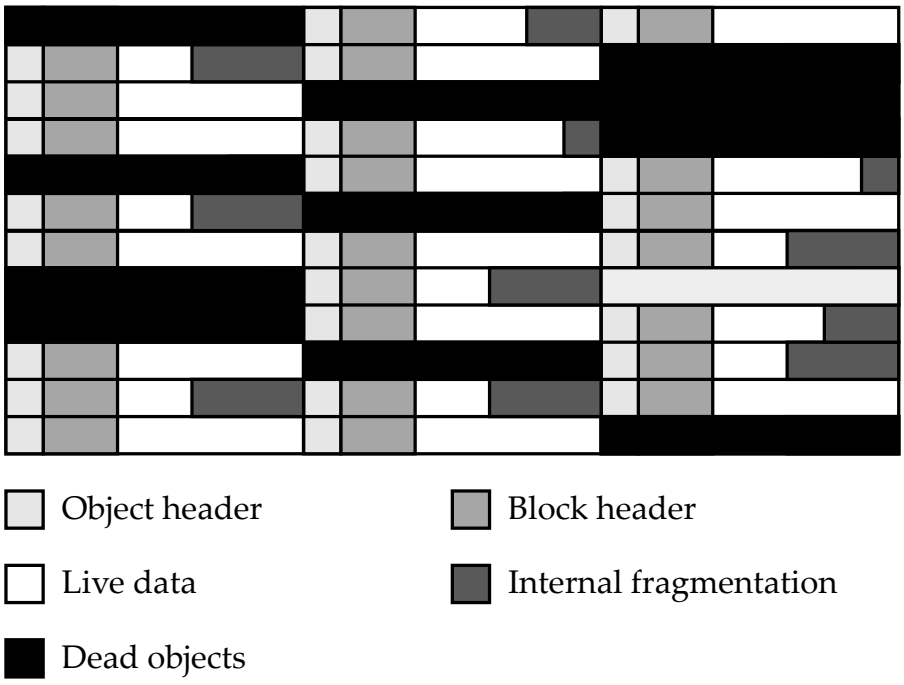


Figure 4.16. Memory layout using RT-Mark-and-Sweep

guaranteeing progress of the GC. The problem of large objects in mark-and-sweep and RTRC is the access time. Using a tree or a linked list increases access costs if objects grow large. Thus, in all cases it is preferable to allocate large objects statically at the initiation of the system, so the GC can ignore them.

4.10.2 Memory Overhead

To be able to compare the memory overhead of RTRC, with the memory overheads of the RT-Mark-and-Sweep, and RT-Copying collectors, we need to investigate the object and heap layout of the techniques.

An example of a typical heap layout of a system that uses RT-Mark-and-Sweep is presented in Figure 4.16. The heap is divided into blocks, which all have a block header (4 bytes.) Each object is constructed from a number of blocks. All objects have an object header (8 bytes.) Half of a block will be lost to internal fragmentation (given that all object sizes are equally probable.) Finally, about $1/3$ of the heap is needed as a buffer to store dead objects while the current GC cycle is completed. This is a fair assumption if the WCET of allocation should be acceptable [Sie02, CB01].

The heap layout of a system using an RT-Copying collector is presented in Figure 4.17. The heap is divided into two sub-heaps, of which only one can be used to store live objects. Each object has an object header (8 bytes),

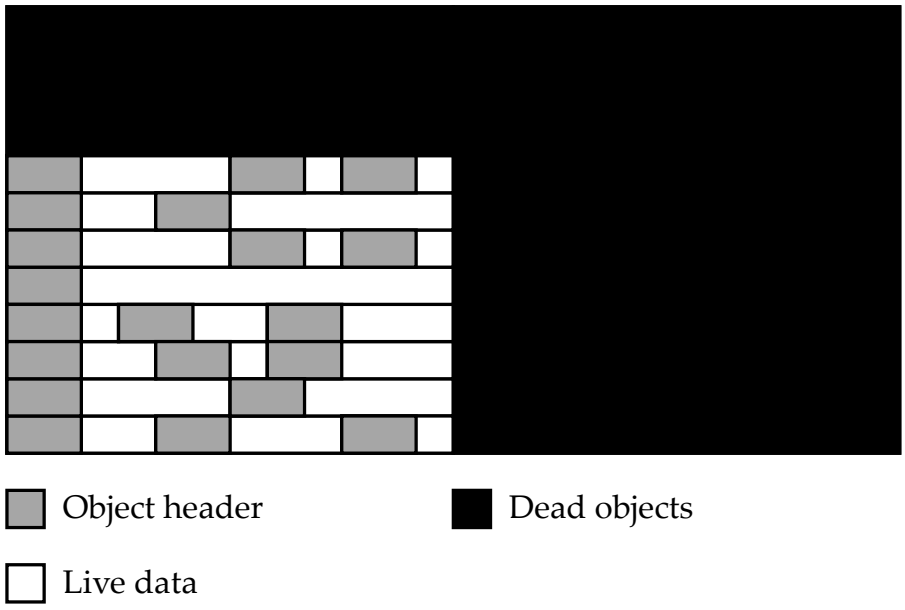


Figure 4.17. Memory layout using RT-Copying

and about $1/3$ of the currently used sub-heap is used as a buffer to store dead objects while the current GC cycle completes, for the same reasons as for RT-Mark-and-Sweep.

To compare the memory overhead of the three techniques, a theoretical model has been set up. The model is presented in Table 4.2. The table shows the formulas for calculating the different kinds of overhead.

The overhead caused by the block header is calculated as the size of the header divided by the size of a block. For RT-Mark-and-Sweep, the value is multiplied by $2/3$, since the other third of the memory contains dead objects. Since RT-Copying does not use blocks there is no overhead associated with it.

The object header causes an overhead of the size of the header divided by the object size (including internal fragmentation). For RT-Mark-and-Sweep and RT-Copying the result is multiplied with the relative amount of memory that is not occupied by dead objects.

The internal fragmentation of RTRC is the number of bytes available in the allocated blocks (the size of the blocks minus the headers) subtracted by the size of the object, and divided by the size of the allocated blocks.

When using RT-Mark-and-Sweep, the internal fragmentation is again multiplied with the relative amount of memory that is not occupied by dead objects. RT-Copying can compacts the objects so no internal fragmentation exists. However, it may be necessary to pad the objects, but here it is assumed that no padding is required.

GC	Block head	Object head	Internal fragmentation	Dead
RTRC	$\frac{BH}{BS}$	$\frac{OH}{BS\,BC}$	$\frac{BC\,(BS-BH)-OH-OS}{BC\,BS}$	0
RT-MS	$\frac{2}{3}\,\frac{BH}{BS}$	$\frac{2}{3}\,\frac{OH}{BS\,BC}$	$\frac{2}{3}\,\frac{BC\,(BS-BH)-OH-OS}{BC\,BS}$	$\frac{1}{3}$
RT-C	0	$\frac{1}{3}\,\frac{OH}{OS}$	0	$\frac{2}{3}$

Table 4.2. Theoretical model of the total memory overhead of RT-RT, RT-Mark-and-Sweep, and RT-Copying. BS is the block size, BH is the block header size, BC is the block count, OS is the object size, and OH is the object header size.

Finally RT-Mark-and-Sweep and RT-Copying needs memory for dead objects. It is assumed that RT-Mark-and-Sweep uses 33 % of the heap and RT-Copying uses 33 % of one sub-heap, plus the unused sub-heap [Sie02, CB01]. The unused sub-heap does not really contain dead objects, but can not store live objects anyway.

The diagrams in Figure 4.18 and 4.19 presents the distribution of the memory overhead using RT-Mark-and-Sweep and RT-Copying using an even distribution of 11 – 33 byte objects. It is clear that large portions of the overhead are used to store dead objects. Since RTRC does not require any memory to store dead objects, it enables the usage of 50 % more memory compared to RT-Mark-and-Sweep and 110 % more than RT-Copying, assuming that there are no dead cyclic data structures.

Finally, the diagrams in Figure 4.20, 4.21, and 4.22 present how much of the heap memory can be used to store actual data using the different techniques and a block size of 16, 32, and 64 bytes. All three diagrams show that the memory utilization in systems using RTRC is superior to the utilization in systems using the previously presented techniques. It is also worth noting that different block sizes do not result in major changes in the memory utilization.

4.11 Summary

The increment operation of standard reference counting just increments a variable, which is an operation with predictable worst-case execution time. However, the decrement operation is potentially recursive. In the worst case a decrement operation can reclaim all objects on the heap, and even if the size of the heap is bounded, it is too long for essentially all real-time applications.

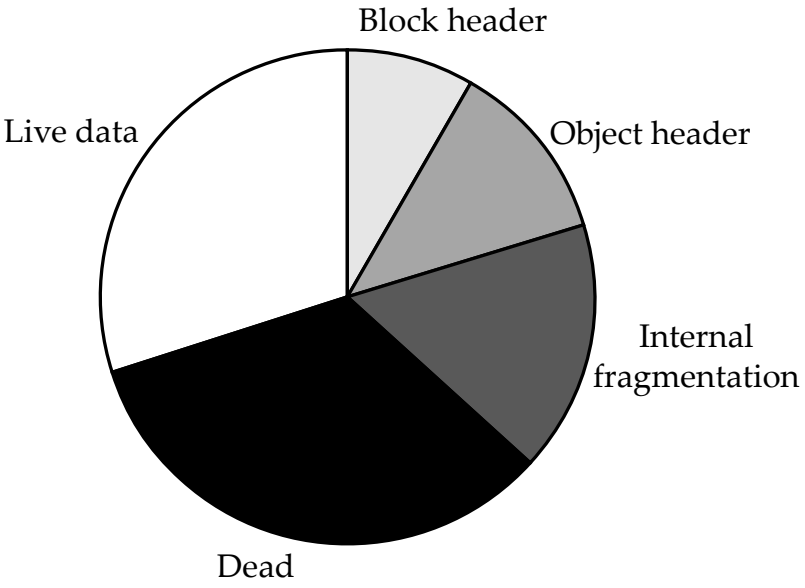


Figure 4.18. Memory overhead of RT-Mark-and-Sweep

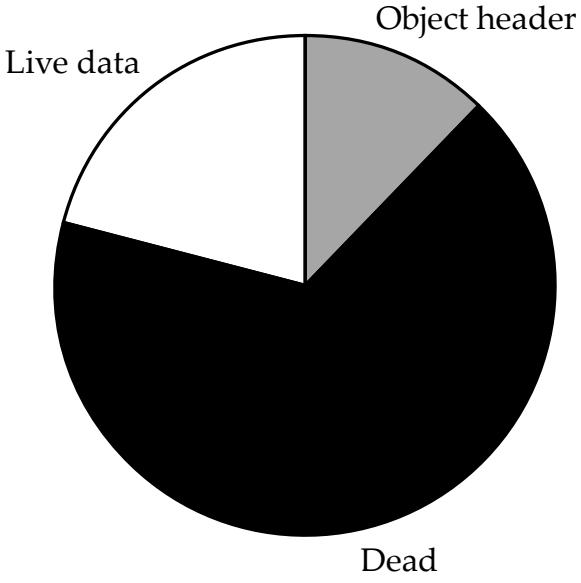


Figure 4.19. Memory overhead of RT-Copying

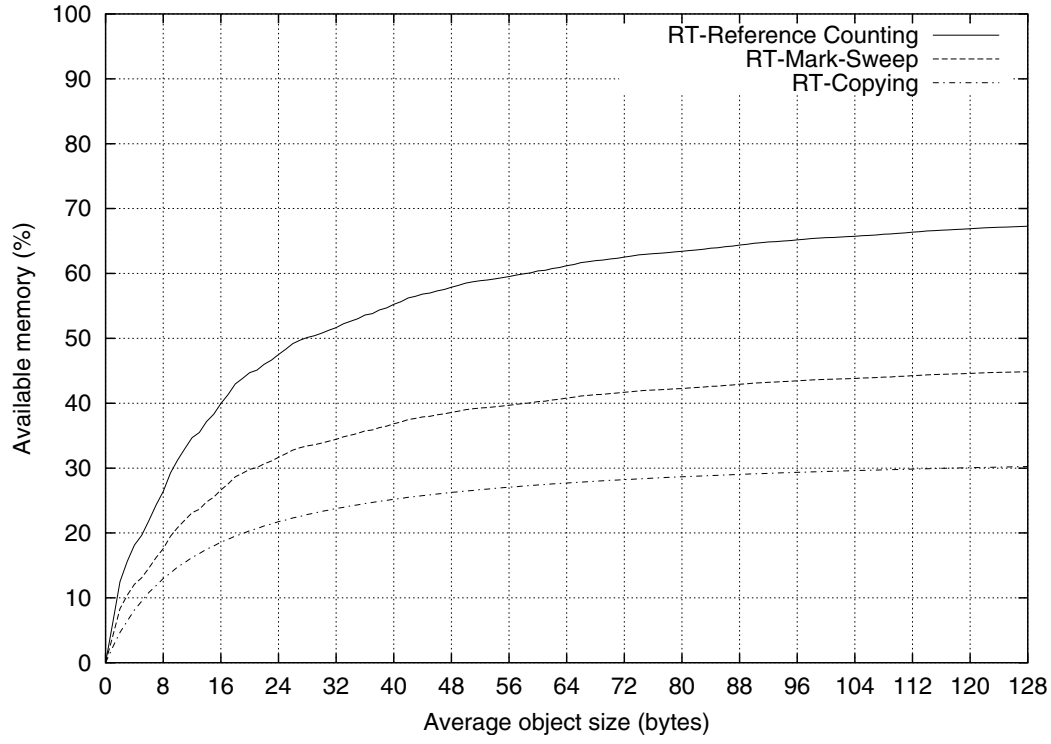


Figure 4.20. Memory overhead comparison diagram using 16 bytes blocks

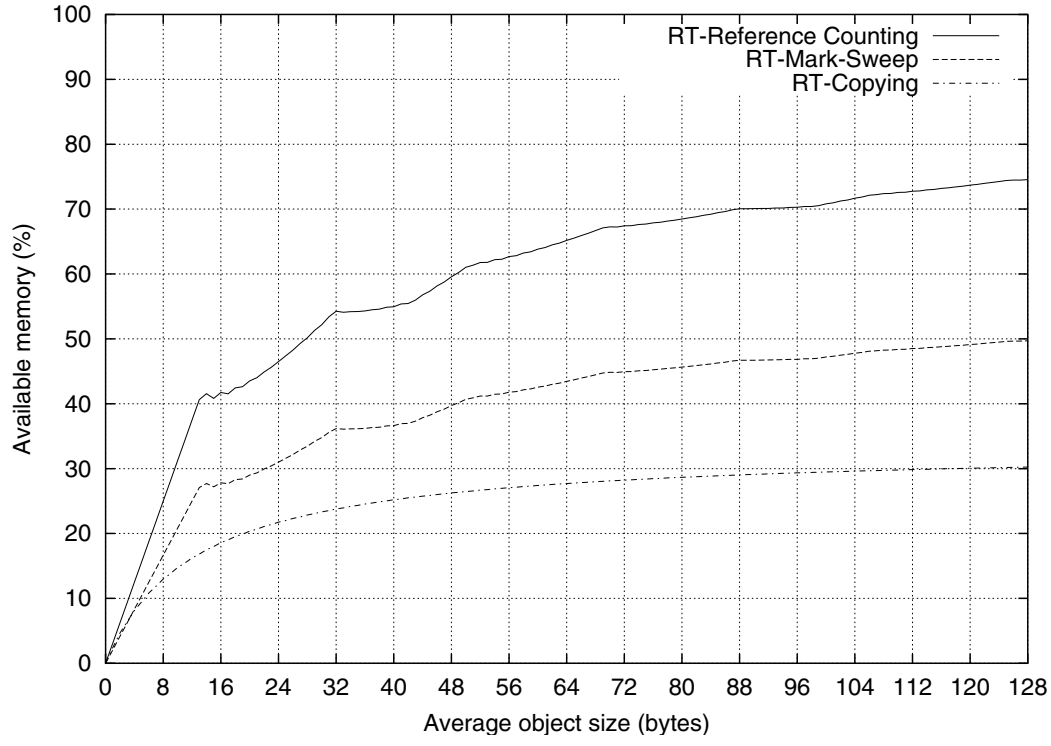


Figure 4.21. Memory overhead comparison diagram using 32 bytes blocks

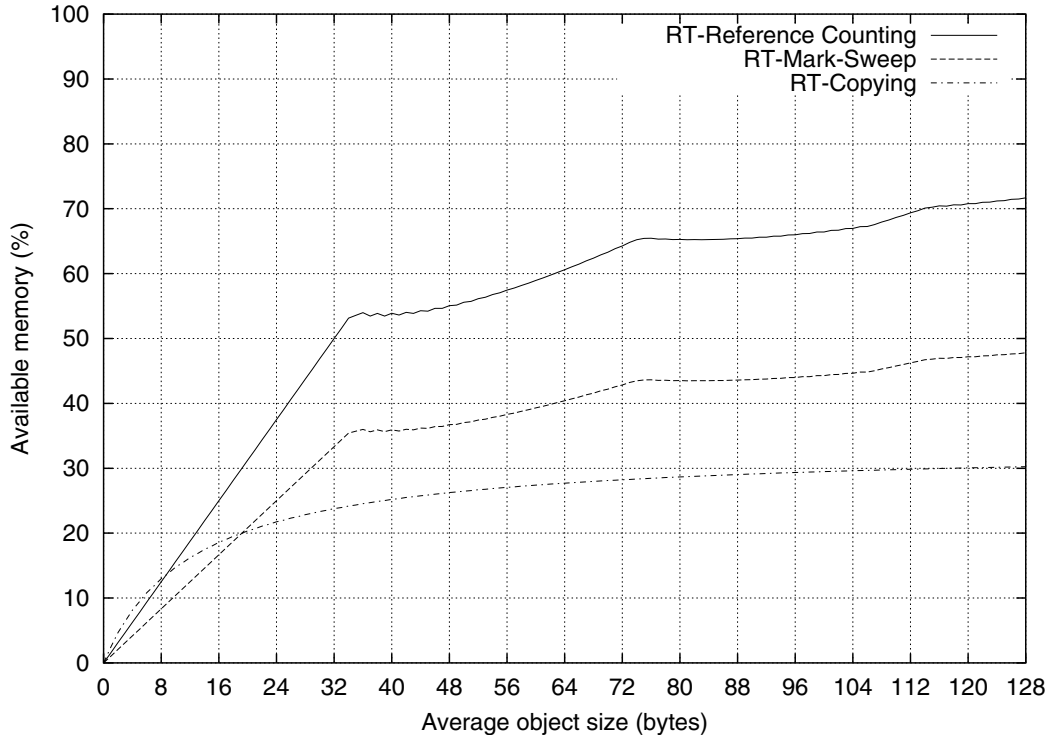


Figure 4.22. Memory overhead comparison diagram using 64 bytes blocks

By using Weizenbaum's technique to eliminate the recursive behavior of the decrement operation [Wei63] the decrement operation becomes constant in time. The objects are added to a to-be-free list instead of decrementing its child references and adding the object to the free list. The child references still need to be decremented, which now is done by the allocator instead. When the allocator needs more memory, objects are taken from the to-be-free list. Before they are reused their child references are decremented.

Weizenbaum's technique assumes that all objects are of equal size, but that is not the case in most systems. However, since our heap is divided into equally sized blocks, the technique can be used on the blocks. To decrease the WCET of allocation, all references in an object are not decremented at once. Instead, only the references in the blocks needed by the new object are decremented. To keep allocation fast, it must be possible to get a single block from the free to-be-free list in constant time.

We now have achieved a system without external fragmentation and with predictable allocation and reference count update operations. It is also guaranteed that all dead blocks are immediately available to new objects. If it is desirable to increase the speed of allocation in high-priority processes, this can be achieved by forcing the free list (which contains blocks whose child references have already been decremented) to contain the number of blocks needed by the high priority process.

Cycles are not handled by this design, but several manual techniques have been presented. These can be used in any combination and should eliminate dead cyclic data structures in most cases. If dead cycles can not be eliminated using the manual techniques, a backup garbage collector can be used. This will, unfortunately, decrease the benefits of RTRC, since all memory will not be reclaimed as soon as possible. However, the system will still require less memory in virtually all cases since all non-cyclic garbage is reclaimed as soon as it dies. If a backup collector is necessary it should have properties that allow it to be smoothly integrated with RTRC. In particular it should be a non-moving incremental garbage collector, such as a mark-and-sweep or a treadmill [Bak94] collector. A disadvantage of using the treadmill as backup is that it requires two pointers in each object, which makes the combination expensive.

Dividing objects into blocks introduces an overhead in both memory usage and execution time. Using a linked list schema, each object needs a pointer to the next block. These pointers need to be traversed when accessing data in blocks other than the first. The number of pointers to traverse to get a field is known at compile-time. Large objects such as arrays should not use a linked schema, because that would make the execution time of an indexing operation a linear function of the size of its argument. Arrays can be stored as trees that make the execution time a logarithmic function of the size of the array. Large objects should preferably be allocated contiguously and statically, so they can be ignored by the collector.

In order to reduce the execution time of reference counting, all reference counting optimizations with predictable behavior can be used. One such technique is Object Ownership, which is presented in Chapter 5.

— Beware of bugs in the above code;
I have only proved it correct, not tried it.

Donald Knuth

Chapter 5

Object Ownership

This chapter presents an optimization that increases the performance of reference counters by eliminating redundant reference count updates. The main factor that slows down a reference counter is memory accesses. In a system without garbage collection, storing a reference is a very fast operation, but when using incremental garbage collection, the GC often need to access the objects involved in the assignment (or some other memory associated with the objects.) When reference counting is used, both the object referred to by the old (left hand side) and new (right hand side) references need to be updated. That means that the objects have to be loaded into the cache and then written back to memory, which is time consuming.

5.1 Basic Idea

The purpose of a reference counter is to reclaim objects as soon as they become unreachable. Thus, it must be possible to decide whether an object is reachable or not. This is equivalent to checking if the reference counter is zero or not, hence the actual number of references to the object is really of no interest as long as it is larger than zero if it is reachable.

The idea of object ownership is to eliminate reference count updates while one reference guarantees that the object is kept alive. The reference that keeps the object alive is said to *own* the object. If another reference temporarily refers to an owned object, it need not update the objects reference counter, since the owning reference will keep the reference counter over zero while it owns the object. The temporary reference must not refer to the object when the owner releases (stops owning) the object, since that would cause the reference counter to be invalid. An example is presented

```
algorithm increment(Integer ii)
    ii.value ← ii.value + 1
end

Integer i ← new Integer(17)
increment(i)
```

Figure 5.1. Redundant reference count update

in Figure 5.1. Before entering the `increment` function, the reference count of the object referred to by `ii` must be greater than zero. The reference count is then incremented by `ii`. When the `increment` function returns, the reference count is decremented since `ii` is released. Now the reference count is the same as immediately before `increment` was called. Thus, the reference count updates caused by `ii` are redundant.

5.2 Owning an Object

This section will define terms that help finding reference count updates that can be eliminated. The approach we use to eliminate redundant reference count updates is based on the time periods in which references refer to objects.

Definition 5.1
The *life time of a reference to an object* is the time from when the reference starts referring to the object until the reference is updated or when it is destroyed (goes out of scope or the containing object is reclaimed).

The life time of a reference to an object should not be confused by the life time of a reference. The life time of the reference can contain several life times of references to different objects.

Definition 5.2
If the life time of a reference to an object is encapsulated in the life time of another reference to the same object, the encapsulated reference is called *inner* and the encapsulating reference is called *outer* reference (see Figure 5.2.)

Lemma 5.1
Reference count updates caused by inner references are redundant.

Proof For each inner reference to an object there must be at least one outer reference to the same object (otherwise there would not be an inner reference.) The outer reference refers to the object during the lifetime of the inner reference, and since all other references that decrement the reference

```
Object a ← new Object()
Object b ← a
...
b ← null
...
```

```
Object a ← new Object()
Object b ← a
...
a ← null
...
```

Figure 5.2. In the example to the left, the `a` reference to the object is an outer reference to the `b` reference. However, in the example to the right does the life span of the `a` reference not encapsulate the lifespan of other reference to the object. Thus, it is not an outer reference.

counter must have incremented it first, the reference count is guaranteed to be greater than zero at all times. Thus, the object will not be prematurely reclaimed.

During the life time of the inner reference, the reference count of the object would first be incremented (when assigning the reference to the object) and then decremented (when assigning a reference to another object or being destroyed). These updates cancel each other, so the reference count is left as it would be if the inner reference would not have performed the updates.

By eliminating the reference count update, the reference count will be invalid while the inner reference refers to the object, but it will not be prematurely reclaimed since the outer reference refers to it. It will neither be forgotten since the reference count becomes valid when the inner reference stops referring to the object. Thus, the reference count update can be eliminated without changing the semantic behavior of the system.

■

The life time of references from the stack can easily be computed since their lifetimes are bounded by their scope. It is much harder to compute the lifetime of references from the heap, i.e. references from other objects. However, reference count updates caused by references from the heap are rare compared to updates caused by references on the stack, so the overhead caused by references from the heap is in general small compared to the overhead caused by references on the stack. Thus, the optimization will still yield a good result if only updates caused by stack references are eliminated.

Lemma 5.2
Reference count updates caused by references from the heap can be performed separately while inner references from the stack are eliminated.

Proof For this proof we count the references from the stack and the heap separately. Neither reference counts can be negative, since there can not be a negative number of references either on the stack or on the heap. This is also true if we allow updates caused by inner references to be eliminated.

Thus, an object can only be reclaimed if both reference counts are zero. It does not matter if the references from the stack are all grouped into one as long as the stack references count becomes zero, only when there are no more references from the stack.

Since the stack's reference count is always larger than zero when updates of inner references are eliminated (since there is an outer reference that is counted), the object will never be reclaimed while the outer reference refer to the object. When the outer reference no longer refers to the object, there can be no inner references referring to the object, so the reference count is valid, and the object is still reclaimed if there are no references from the stack or the heap.

■

Finally we can define which references own objects and prove that references from the stack to owned objects can be eliminated.

Definition 5.3
A reference, R, to object O *owns* O if no outer reference to R owns it, and if all other references from the stack that refers to O while R does are inner references to R.

Theorem 5.1
References from the stack to owned objects are redundant and can be eliminated while non-local references are counted as in the original reference counting technique.

Proof Since all references from the stack to an owned object are inner references to the reference that owns the object (Definition 5.3) their reference count updates are redundant (Lemma 5.1 and 5.2). Thus, references from the stack to owned objects need not be counted.

■

5.3 Static Analysis

By limiting the optimization to references from the stack, the analysis becomes simpler. Within a method, simple def-use analysis can be used to find inner and outer references. However, this can not eliminate the reference count updates that occur when references are passed as arguments to methods.

When a reference to an owned object is passed to as an argument, all reference updates to this object caused by references on the stack can be eliminated. Thus, all references to that object from the stack within all directly or indirectly called methods are inner references since their lifetimes are bounded by the call.

The only requirement to own an object that is passed as an argument is that the reference is not modified during the call. The only way a reference in a higher stack frame can be modified from a called method is if a reference to the reference is available in the current stack frame (or on the heap.) Many modern languages, such as Java, do not allow such double indirections, but if it is allowed it can easily be detected by prohibiting references that store or pass a reference to itself from owning objects.

Since it is very common to pass references from the current stack frame as parameters, many objects passed to methods will be owned, thus the optimization will eliminate many reference count updates in these objects. Many cases where a reference that is not stored in the current stack frame is passed to methods can be rewritten to store the reference in a variable first without loosing performance. One should be careful if the object may be reclaimed in a called method, since keeping a local reference to such an object will keep it alive, and thus increase the memory usage of the application. However, eliminating all reference count updates to the object may be worth loosing some memory.

To be able to tell that an object passed by reference is owned, all references passed to that parameter must be owned, so all calls must be examined. Note that it is not important to know which reference owns the object. It is enough to know that some reference does.

5.3.1 Supporting Separate Compilation

If inter-procedural analysis is regarded too expensive or if separate compilation is a requirement, it is not possible to know if objects passed as parameters to some method are owned or not. Even if inter-procedural analysis is used, a passed object may sometimes be owned and sometimes not. The conservative approach is to treat all unknown objects as not owned. However, by adding more runtime information these objects can also benefit from the object ownership optimization.

By adding a boolean field in the object header that tells whether an object is owned or not would make it possible to check whether the object is owned or not. Unfortunately, this requires a memory access, but if the object is owned it is not written back to memory (since it is not updated), thus the performance increases. If a boolean field is used, it must be known when an object becomes owned, so that it can be released when that reference is no longer referring to it. In many cases this will be very difficult to tell.

A solution is to exchange the boolean field for a reference referring to the owning reference. The field is set to null when no reference owns the object, allowing the increment and decrement operations to check it and skip updating the reference counter. The updated increment and decrement operations are presented in Figure 5.3 with the operations that take and release the ownership of an object.

```
algorithm take_ownership(Reference reference, Object obj)
    if obj.owner = null then
        obj.owner ← adressOf(reference)
    end if
end

algorithm release_ownership(Reference reference, Object obj)
    if obj.owner = adressOf(reference) then
        obj.owner ← null
    end if
end

algorithm rcinc(Object obj)
    if obj ≠ null and then obj.owner = null then
        obj.rc ← obj.rc + 1
    end if
end

algorithm rcdec(Object obj)
    if obj ≠ null and then obj.owner = null then
        obj.rc ← obj.rc - 1
        if obj.rc = 0 then
            freelist.add(obj)
        end if
    end if
end

algorithm write-barrier(Object in out lhs, Object rhs)
    rcinc(rhs)
    rcdec(lhs)
    lhs ← rhs
end
```

Figure 5.3. Optimized Reference Counting

5.4 Benchmarks

Even though the optimization has not been implemented in a compiler, measurements of the updated increment and decrement operations have been performed using the test program in Appendix B. The test program uses the updated write-barrier (as presented in Figure 5.3) by doing simulated assignment of references stored in an array. The array in the test contained 1 000 000 objects of 12 – 16 bytes each (depending on whether the extra references was added to the header.) Thus, it is large enough to keep the data out of the cache. The test program was compiled in four versions:

- with no write-barrier
- with plain RTRC
- with object ownership and all objects owned
- with object ownership and no objects owned

All versions were executed with four difference uses:

- without accessing the objects
- accessing the left-hand-side
- accessing the right-hand-side
- accessing both objects

These tests were run on three configurations:

Config 1 Mobile Pentium III, 700 MHz, 192 MB, running on battery

Config 2 Mobile Pentium III, 700 MHz, 192 MB, running on AC

Config 3 AMD Athlon XP 1800+, 768 MB

Every program was executed five times, and the shortest execution time was used in the calculations. The shortest execution time was chosen since it was the execution with least disturbance. The calculations were also made using the average and maximum execution time, and the results were practically the same.

To evaluate the impact of the optimization, the execution time of the write-barrier using plain RTRC was compared to the execution time in the two cases using object ownership. The execution time of the write-barrier was calculated by subtracting the execution time of the version without any barrier code. The results are presented in Figure 5.4 which presents the increase in execution time performance when objects were owned, and in Figure 5.5 which presents the impact when objects were not owned.

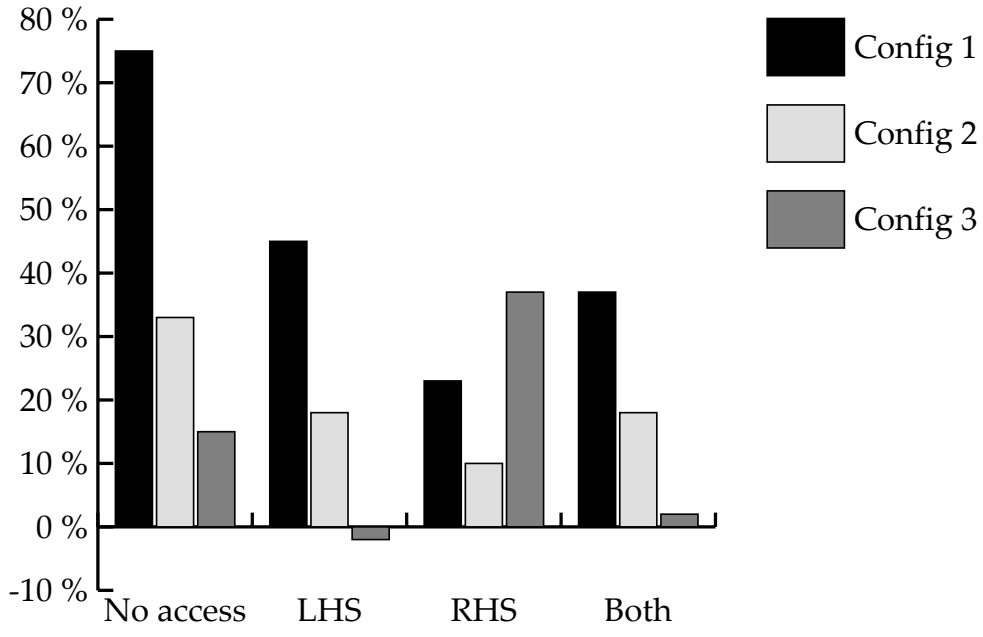


Figure 5.4. Execution time performance of the write-barrier using Object Ownership when objects are owned

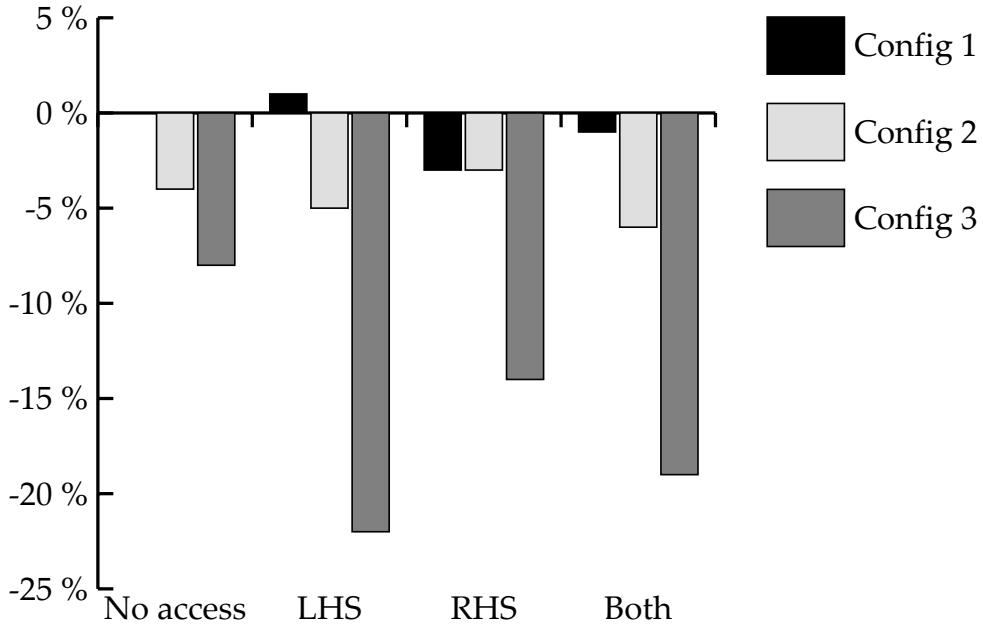


Figure 5.5. Execution time performance of the write-barrier using Object Ownership when objects are not owned

Note that these are measurements of the performance of the write-barrier. It does not say anything of how often the optimization can be applied. Note also that if object ownership is used to eliminate the write-barrier, the overhead is also completely eliminated.

When running the laptop (configuration 1) on battery, it is in a power saving mode, which among other things decrease the frequency of the CPU clock. The performance of the write-barrier increase with 23 % – 75 % with an average of 45 %. When using external power (configuration 2), the computer goes out of the power savings mode, and the performance increase with 10 % – 33 %, compared to the plain RTRC version, with an average of 20 %. Using the desktop computer (configuration 3), the performance increase -2 % – 37 % with an average of 13 %. The optimization clearly depends on how the system uses the cache. These tests indicate that slower systems may gain more on the optimization.

When running the program when the objects are not owned, the optimization adds a check in the write-barrier. Since the objects are not owned, the write-barrier is executed as in the plain RTRC implementation. Still, configuration 1 practically does not show any decrease in performance (-3 % – 1 % increase.) Configuration 2 shows a slight decrease. The performance decreases 3 % – 6 %. However, configuration 3 show a decrease of execution time performance of 8 % – 22 % with an average of 16 %.

Thus, some systems have much to gain from using this approach. Configuration 1 and 2 barley loose any performance when the objects are not owned and the execution time performance increase by as much as 75 %. Configuration 3 may gain as much as 33 %, but when the objects are not owned it may also loose 19 %. Thus, in that case it should only be used if most objects are owned.

5.5 Extensions

Object ownership can be very useful as it is, but there are still some possibilities for extending the technique.

5.5.1 Explicit Freeing

If the reference that stores a newly allocated object owns the reference and no references to the object are stored on the heap, i.e. in a field of another object, the object can be explicitly reclaimed when the reference releases the object since no other object on the stack can refer to it (then the first reference could not have owned the object.) When this can be applied, the reference counter need not perform any extra work with this object. The only GC overhead associated with the object is the memory occupied by the reference counter. This idea is further developed in the next chapter.

5.5.2 Overlapping References

Sometimes a reference can not take ownership of an object because another reference starts referring to the object after the first one and still refers to it when the first reference releases the object, i.e. the life times of the references to the object overlaps, but one is not enclosed in the other. A simple solution is to create a third reference that is an outer reference to both references. This reference owns the object and releases it when the last reference is updated or goes out of scope. The third reference need not actually be created, the ownership of the reference may be handed over to the other reference. This extension would make the optimization applicable to more code.

5.5.3 Supporting Other GC Techniques

An interesting idea is to use the object ownership technique for other incremental garbage collection techniques. What actually happens is that all overhead caused by references from the stack to owned objects can be ignored by the garbage collector. This ought to be useful information for other garbage collectors as well. All barrier code can be eliminated when an object is known to be owned. This should increase execution time and decrease code size for most systems.

5.6 Summary

This chapter introduced an optimization technique for reference counting. The technique eliminates reference count updates that are redundant because another reference is known to keep the object alive while the reference refers to the object. Since references from the heap are hard to analyze, only updates by references from the stack are eliminated. References from the heap can still refer to the same object without corrupting the analysis.

We also present a small benchmark of the version of object ownership that adds a field in the object header and extra runtime checks. Even though extra code is added, the optimization increase the execution time performance of the write-barrier with up to 75 %. This version is used when it is not known whether an parameter is owned or not. If all of the system is available at compile time, the optimization can eliminate the write-barrier completely when objects are owned, and thus completely eliminate the execution time overhead caused by the write-barrier.

— Time is an illusion. Lunchtime doubly so.

Ford Prefect

Chapter 6

Static Garbage Collection

Garbage collection work is normally performed during runtime, however static analysis can be used to relieve the runtime GC of work or even make it redundant. Such analysis, called static garbage collection, generally increases the performance and predictability of runtime GCs of all systems. Performance can be gained since the runtime garbage collector needs to pay less attention to objects that can be handled statically. Objects can sometimes be allocated globally or on the stack to decrease execution time of allocating and reclaiming objects. Predictability is gained since free operations become explicit in the application, making the memory usage calculations easier. If all objects are handled by the static analysis, and the worst case execution time can be calculated, the worst case memory usage can often also be calculated, a task which can be extremely difficult when using a runtime garbage collector.

All analysis presented in this chapter assumes that there is no dynamic loading of classes and that introspection is not used. These are normal restrictions for hard real-time systems, where applications execution behavior must be predictable. It is still possible to use dynamic loading if the complete system is re-analyzed, and all classes that are modified by the developers or by the optimizers are uploaded.

6.1 Overview

The static garbage collector presented here consists of two parts. First the application is analyzed using an extended escape analysis. Escape analysis [PG92] is used to examine whether references escape methods, i.e. if objects live longer than a method's stack frame. This information is commonly used to allocate objects on the stack if possible. Here, escape analysis has been extended to handle cases even when objects lives extend be-

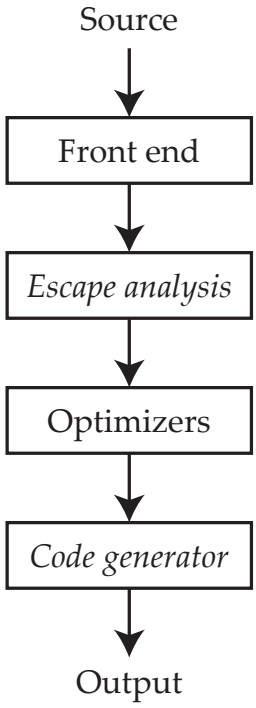


Figure 6.1. An overview of a compiler that uses static garbage collection. To include the static GC in a compiler, the escape analysis must be introduced and the code generator needs to be extended.

yond the life time of the method in which they are allocated. The result from the escape analysis is used by the code generator to generate optimized code by inserting explicit free instructions, and possibly by modifying allocation instructions (see Figure 6.1.)

The escape analysis is divided into several phases. First a combined data flow and call graph is constructed from the intermediate representation of the compiler. The call graph is then converted into a call tree, and the data flow graph is updated to fit the call tree. Finally the data flow from the allocations is traversed to find out where objects can be explicitly reclaimed. The output of the analysis is the minimum stack depth from which each object is accessed. An object can be reclaimed when control returns to a stack frame above the level from where the object can be accessed. The analysis is further discussed in Section 6.3.

The code generator is extended to optimize allocating and freeing objects using the result from the escape analysis. Allocations can be turned into stack allocations, or if it is possible they can refer to statically allocated objects. It is also possible to insert explicit free instructions in the code to handle some objects. The extensions of the code generator are further discussed in Section 6.4.

```
class SGCExample {
    int i;

    void f() {
        SGCExample obj = new SGCExample();

        obj.i = 17;

        // and something interesting
        // ...
    }
}
```

Figure 6.2. Java program that is used to present program transformations

6.2 Optimizing Memory Management

The primary goal of a static garbage collector is to relieve the runtime garbage collector of work. Consider all allocation instructions in a system. The objects that these allocations create can all be handled by a runtime garbage collector, but it is also possible to free the memory using explicit free instructions (if the runtime system supports it.) However, to do this automatically is currently not within reach. Still, it is possible to trace the data flow of references to find methods from which objects do not escape. These objects can be reclaimed when the execution of that method has finished. When an object does not escape the method it is allocated in, it can even be allocated on the runtime stack, which further improves allocation and free performance. If a method is not recursive, there can only exist one instance of its activation record on the stack. Thus, there can be at most one instance of the object at any given time. Therefore, the object can be allocated statically, which practically eliminates the need for allocation and freeing completely. In a multi-threaded environment where the method that allocates the object can be invoked by different threads, one object per thread needs to be allocated.

The following sections describe different program transformations that can be made as the result of the escape analysis. The transformations are presented as a Java program translated into C. The transformation could not be Java-to-Java since Java lacks some features that are required. Still, the transformations can be used in both Java-to-native compilers and Java virtual machines by updating the compiler and/or virtual machine. The Java program is presented in Figure 6.2 and the direct translation into C is presented in Figure 6.3.

```
typedef struct Static {
  // Type and GC attributes stripped out
  int i;
} Static;

void Static_f(Static *this) {
  // Allocate the object
  Static *obj = malloc(sizeof(Static));

  // Initialize and call constructor
  Static_initialize(obj);

  obj->i = 17;

  // and something interesting
}
```

Figure 6.3. Unoptimized C translation

6.2.1 Static Allocation

This section presents an approach to handle allocations of objects that die before the next execution of the allocation instruction, i.e. allocations that can share a single instance of an object. These objects can be allocated statically as global (or static) objects.

These allocations can be replaced by a reference to the global object and a call to an initialization method (including a call to a constructor.) No free instruction is required since the same instance is reused at the next invocation. An example of such a program transformation is presented in Figure 6.4.

If an allocation instruction is part of a loop, it can create multiple objects within the same method invocation. Then it is impossible to refer to a single instance for all allocations (if the created object does not die before the next one is created.) Such allocations can be handled by creating an array of objects statically. The array must be large enough to handle all objects that are created in the loop. Thus, the loop should have a fixed upper bound on the number of iterations. The upper bound should also be relatively tight to the average case, otherwise memory resources are wasted. The program transformation must assure that all allocations return distinct objects. To present this transformation, the Java program in Figure 6.5 has been transformed and translated to C in Figure 6.6.

Typical examples of allocations that are handled by allocating static objects are allocations of constant objects. Allocations of temporary objects in non-recursive methods commonly fit into this category as well.

```
// Allocate the object staticaly
SGCExample SGCExample_f__obj;

void SGCExample_f(SGCExample *this) {
  SGCExample *obj;

  // Point to static object
  obj = &SGCExample_f__obj;

  // Initialize and call constructor
  SGCExample_initialize(obj);

  obj->i = 17;

  // and something interesting
  // ...
}
```

Figure 6.4. C version using static allocation

```
class SGCMultiple {
    void f() {
        SGCExample[] objs = new SGCExample[10];

        for (int i = 0; i < objs.length; i++) {
            objs[i] = new SGCExample();
        }

        // Do something interesting
        // ...
    }
}
```

Figure 6.5. Java method that allocates multiple objects per invocation

A conservative approach to find such allocations is to search for allocations that meet these requirements:

- The allocation must be in a non-recursive method
- The method must not be invoked by multiple threads
- The allocated object must not escape the method

Many more allocations could be handled using this approach. However, the complexity of the analysis increases if the objects are allowed to escape the method. Most of the cases that can be handled by this approach, but does not meet the requirements above, can be handled statically by some of the approaches that follow.

```
SGCExample *SGCMultiple_f__objs[10];
SGCExample SGCMultiple_f__objs_array[10];

void SGCMultiple_f(SGCMultiple *this) {
  SGCExample *(*objs)[];
  int i;

  objs = &SGCMultiple_f__objs;
  Array_initialize(objs, 10);

  for (i = 0; i < Array_length(objs); i++) {
    (*objs)[i] = &SGCMultiple_f__objs_array[i];
    SGCExample_initialize((*objs)[i]);
  }

  // Do something interesting
  // ...
}
```

Figure 6.6. C version of the program in Figure 6.5 using static allocation

This approach requires the runtime system to be augmented with a method that initializes the object. It must also be possible to call a constructor. If all constructors fully initialize objects, there is no need for a separate initialization method.

6.2.2 Thread Local Allocation

Allocations that pass the above analysis except that the method that contains the allocation is called from more than one thread can be allocated in thread local memory.

Several approaches to program transformations are possible for these allocations. Some runtime environments, e.g. the Java 2 platform, support thread local variables directly. A more general approach that uses a map that is indexed by a thread ID to get the thread local object is presented in Figure 6.7. If multiple objects are allocated, the same approach as described for static allocation can be used.

This approach does not put any requirements on the runtime system other than the once presented above.

6.2.3 Stack Allocation

Allocations that meets the requirements for the "thread local" category, except that the method is recursive can be allocated on the runtime stack if the runtime systems supports it.

```
// A set of objects (one for every thread). When
// a thread is created it must also add an object
// to this set. Alternatively checks must be done
// to every time an object is "allocated", and an
// object is added if it does not exist
set _thread_SGCExample_f__obj;

void SGCExample_f(SGCExample *this) {
  SGCExample *obj;

  // Get the thread local version of obj
  obj = set_get(_thread_SGCExample_f__obj, thread_id());

  // Initialize and call constructor
  SGCExample_initialize(obj);

  obj->i = 17;

  // and something interesting
  // ...
}
```

Figure 6.7. C version using thread local objects

```
void SGCExample_f(SGCExample *this) {
  // Allocate memory in stack frame
  SGCExample _obj;

  SGCExample *obj;

  // Point to the stacked object
  obj = &_obj;

  // Initialize and call constructor
  SGCExample_initialize(obj);

  obj->i = 17;

  // and something interesting
  // ...
}
```

Figure 6.8. C version using a local variable

```
void SGCExample_f(SGCExample *this) {
  SGCExample *obj;

  // Allocate object in stack frame
  obj = alloca(sizeof(SGCExample));

  // Initialize and call constructor
  SGCExample_initialize(obj);

  obj->i = 17;

  // and something interesting
  // ...
}
```

Figure 6.9. C version using `alloca()` function

The transformation changes the allocation statement to allocate the object on the stack instead of on the heap. This can be achieved either by making it a local variable, as in Figure 6.8, or using an allocation function that allocates memory from the stack, as in Figure 6.9 using the `alloca()` function in available in many C runtime libraries. In either case no free instruction is necessary since all memory that is allocated is allocated on the stack is implicitly freed when the method returns.

A local array can be used to store the objects if a fixed number of objects are allocated. However, if the number of allocations varies from time to time, valuable memory resources may be wasted. If this is a problem, the `alloca()` approach can be used instead.

This approach requires the runtime system to support allocation on the runtime stack, and that the objects allocated there can be initialized as required before.

6.2.4 Explicit Freeing

In theory, all allocations can be handled using explicit free instructions, as is the case when using manual memory management. However, we are far from handling all allocations by automatically inserting free instructions. This problem is extremely hard if memory leaks are not accepted.

Our approach to the problem is to trace the data flow from the allocations to all their uses using the escape analysis described below. The output from the analysis is the minimum stack depth from which each allocated object is accessed.

The program translation can either insert explicit free instructions to free objects, but this can become complicated. A simpler and still efficient solution is to associate a set with each stack level. When an object is allo-

```
// An array of sets. One for every stack level.
set _free_sets[MAX_STACK_DEPTH];

// Variable that holds the current stack depth.
// It is updated when entering and leaving methods
unsigned int _stack_depth;

void SGCExample_f(SGCExample *this) {
  SGCExample *obj;

  _stack_depth++;

  // Allocate object on the heap
  obj = malloc(sizeof(SGCExample));

  // Schedule it to be freed when leaving
  // this stack level
  set_add(_free_sets[_stack_depth]);

  // Initialize and call constructor
  SGCExample_initialize(obj);

  obj->i = 17;

  // and something interesting
  // ...

  set_forall(_free_sets[_stack_depth], free);
  _stack_depth--;
}
```

Figure 6.10. C version of using explicit free

cated, it is also added to the set that is associated with the stack level where it can be reclaimed. When execution leaves a method, all objects associated with its stack level are freed. Given the output of the escape analysis, this code is trivial to output. There is a small overhead in memory usage to keep the sets and a small overhead in execution time to handle the sets. The transformation is presented in Figure 6.10. This solution is further discussed in Section 6.4.

6.2.5 Variable Sized Objects

To statically allocate objects that can vary in size, such as arrays, may cause problems if the sizes vary much. One possibility is to allocate the largest possible object (if this is known), but this can lead to waste of too much

memory. The problem arises only when using static allocation or using local variables to store objects (since most languages do not support variable sized local variables.) The solution is simply to use one of the other approaches to handle these objects. If an `alloca()` function is available this can be used to store the object in the stack frame, otherwise explicit freeing can be used.

6.2.6 An Example

The example in Figure 6.11 and 6.12 shows how a Java program with three allocations can be transformed into C++ code where all allocations are handled statically.

The method `compare()` compares the sum of an array of complex numbers with a complex number that is passed as two doubles. To sum up the array the method `add()` is called. This method creates two objects. The first one is the object that is passed as the result from the method and the second is a temporary object. The third allocation is in the `compare()` method, and it creates a temporary object that holds the complex number which is compared to the sum returned from `add()`.

In the transformed program, the first allocation (of the result of the `add()` method) is allocated on the heap and reclaimed explicitly in the `compare()` method. The second allocation is allocated on the stack, since it is part of a recursive method, and does not escape it. Finally, the complex number that is allocated in the `compare()` method is allocated statically, since it does not escape and it is not part of a recursive method. To save memory, it can also be allocated on the stack.

6.3 Extended Escape Analysis

The goal of the escape analysis is to find the earliest point in the application where objects can be reclaimed. This is achieved using an inter-procedural data flow analysis that searches for the minimum stack level from which each object is accessed.

6.3.1 The Data Flow Graph

The graph represents both the data flow and the method calls of an application. The nodes represent sources of data and methods, and the edges represent the data flow and method calls. There are eight kinds of nodes:

- Class nodes represent classes which are global and may contain fields
- Null nodes represent null values (only one is needed)
- Variable nodes represent local variables and actual arguments

```
class Complex {
    double r, i;

    Complex(double r, double i) {
        this.r = r;
        this.i = i;
    }
}

Complex add(double[] v, int i) {
    if (i == v.length) {
        return new Complex(0, 0);
    }
    else {
        Complex c = new Complex(v[i], v[i+1]);
        Complex tail = add(v, i + 2);
        tail.r += c.r; tail.i += c.i;

        return tail;
    }
}

boolean compare(double[] v, double r, double i) {
    Complex c1 = new Complex(r, i);
    Complex c2 = add(v, 0);

    return c1.r == c2.r & c1.i == c2.i;
}
```

Figure 6.11. Java program with three allocation statements

- Method nodes represent methods (call graph) and their return values (data flow graph)
- String nodes represent string constants
- Field nodes represent fields (static or not)
- New nodes represent allocations
- Throw nodes represent thrown exceptions

These nodes are connected using five kinds of edges:

- Send-to edges represent assignments
- Pass-to edges represent argument passing

```
struct Complex {
  double r, i;
  Complex(double r = 0, double i = 0) { initialize(r, i); }
  Complex *initialize(double r, double i) {
    this->r = r; this->i = i;
    return this;
  }
};

Complex *add(vector<double> v, int i) {
  if (i == v.size()) {
    // Heap allocation that is freed above
    return new Complex(0, 0);
  }
  else {
    // Stack allocation
    Complex c(v[i], v[i+1]);
    Complex *tail = add(v, i + 2);
    tail->r += c.r; tail->i += c.i;

    return tail;
  }
}

Complex compare__c1;

bool compare(vector<double> v, double r, double i) {
  // Static allocation
  Complex *c1 = compare__c1.initialize(r, i);
  Complex *c2 = add(v, 0);
  bool returnValue = c1->r == c2->r && c1->i == c2->i;

  // Explicit free of object returned from add()
  delete c2;

  return returnValue;
}
```

Figure 6.12. Eliminating the use of a runtime GC in Figure 6.11

```
class SGCGraph {
    SGCGraph field;

    static void f() {
        SGCGraph obj1 = new SGCGraph();
        SGCGraph obj2 = new SGCGraph();

        obj2 = g(0, obj1, obj2);
    }

    static SGCGraph g(int i, SGCGraph p1, SGCGraph p2) {
        p2.field = new SGCGraph();

        if (i == 0) return p1;
        else if (i == 1) return p2;
        else return new SGCGraqh();
    }
}
```

Figure 6.13. The Java program that is presented as a graph in Figure 6.14

- Return-to edges represent a value being returned
- Field edges connects fields to their objects (bidirectional)
- Call edges connects methods with the methods they call (bidirectional)

The graph of the Java program in Figure 6.13 is presented in Figure 6.14. Note that variables and parameters that do not contain references are ignored by the analysis and are not included in the graph.

6.3.2 Building the Data Flow Graph

The graph is constructed in the first phase of the escape analysis. It is done by traversing all classes, all methods, all their basic blocks, and all instructions. This procedure is specific for each compiler. The creation procedure for the Jamaica VM builder is presented in Section 7.3.2.

6.3.3 Converting the Call Graph into a Tree

The current design requires that the call graph is converted into a call tree to be able to analyze each invocation of each method separately. The reason for this is to separate calling contexts and to make it possible to associate a unique stack depth to each invocation (except for recursive calls.) Converting is a major disadvantage since it enlarges the input to the analysis

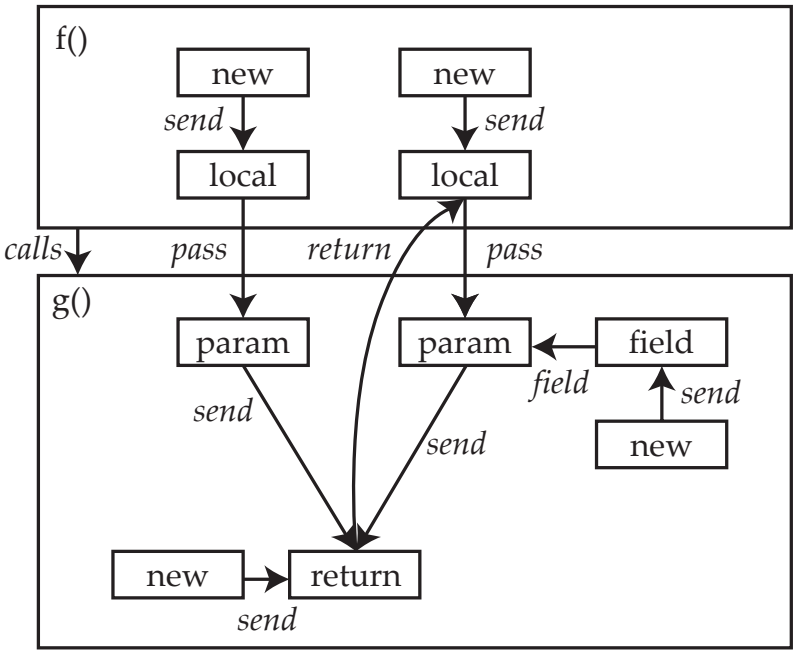


Figure 6.14. The graph of the Java program in Figure 6.13.

and thus requires more memory and CPU resources. However, the analysis phase can be redesigned to use a graph instead of a tree. This is further discussed in Section 6.6.2.

The conversion is started by calling the `makeTree()` method, which is presented in Figure 6.15, on the main-method node. This method converts the call graph into a call tree (with back edges for recursive calls) in a bottom-up fashion. To detect recursive calls, the path of the called methods from the main method to the caller of the currently analyzed method is passed as an argument to the `makeTree()` method. This path is checked to see if it already contains the current method node. If it does, the call is recursive, and a counter that keeps track of the number of recursive calls to each method node is incremented, all methods in the recursion are marked as such, and the current node is returned. If the node is not recursive, the nodes level is set to the length of the path, and the node is added to the end of the path.

The next step ensures that all call edges from the node refers to trees by calling `makeTree()` recursively on each of them. All call-edges are replaced by edges to the resulting trees that are returned from the recursive calls to `makeTree()`.

To guarantee that the returned method has exactly one non-recursive caller, the method node is duplicated if the number of non-recursive callers is greater than zero. When a method is duplicated, a deep copy is performed.

The `deepCopy()` method also takes a list of method nodes as its argument to detect recursive calls. If a recursive call is detected, the first instance of the method node is returned to generate a call to the correct instance of the method (the one that is an ancestor of the called method.) If the method is not recursive, the method is cloned. When a method is cloned, all directly or indirectly called methods are recursively cloned and calls and callers edges are added.

In the next step, all local nodes (arguments, variables, allocations, etc.) are copied to the new copy of the method. However, the edges of the cloned local nodes still target the local nodes of the original method. Therefore, the local nodes in the cloned method are rewired. When a node is rewired, all edges to other local nodes are re-targeted to target the corresponding node in the cloned method. This is achieved using the local nodes' unique indexes in the list that contains all local nodes in each method node. Finally, the cloned node is returned.

Now, the graph has been converted into a call tree (with recursive calls allowed), but the data flow between the methods is still corrupted. This is fixed by the `relinkCalls()` method after the call tree has been created. This phase filters out and re-targets edges so that values only can be returned to calling methods, and parameters only can be passed to called methods. As a special case a return value can be passed to a sibling method, i.e. a method that is called by the method that calls the method that returns the value. The `relinkCalls()` method is called recursively on all non-recursively called method nodes.

6.3.4 The Escape Analysis

The analysis starts by searching for allocations in the main method of the application using the `markAllocations()` method as presented in Figure 6.16. When an allocation is found, its data flow is traversed as described below. When all allocations in a method have been processed, all methods that are called from it are analyzed to find allocations. This continues recursively until all called methods have been analyzed.

Next all started threads are analyzed by analyzing their start methods (`Thread.start()` in Java). All objects that are used by multiple threads are handled by the runtime GC, since this analysis can not decide the execution order in different threads.

Figure 6.17 presents the `markDataFlow()` method, which analyzes the data flow of general nodes. The input to the method is the current stack depth, and the output is the minimum level from which it has been found to be accessed. The analysis of a node recursively follows all send-to edges (assignments) and passed-to edges (parameter passing). When following a pass-to edge, the passed stack depth parameter is increased by one, since it is passed to a called method. The result of analyzing a node

```
algorithm makeTree(MethodNode this, List path) returns MethodNode
    MethodNode recursive ← null
    foreach n in path do
        if recursive ≠ null or n = this then
            recursive ← n
            n.recursiveCalls ← n.recursiveCalls + 1
        end if

        if recursive ≠ null then
            n.recursive ← true
        end if
    end loop

    if recursive ≠ null then
        return recursive
    end if

    level ← size(path)
    addLast(path, this)

    Set backup ← this.calls
    this.calls ← new Set()
    foreach n in backup do
        remove(n.callers, this)
        addCall(this, makeTree(n, path))
    end loop

    removeLast(path, this)
    this.level ← -1

    if this.callers.size() > this.recursiveCalls then
        return deepCopy(this, path)
    else
        return this
    end if
end
```

Figure 6.15. Converting the graph into a tree

```
algorithm markAllocations(MethodNode this)
    foreach l in this.locals do
        if l instanceof NewNode then
            NewNode n ← l
            n.accLevel ← markDataFlow(n, this.level)
        end if
    end loop
    foreach m in this.calls do
        if m.level > level then
            markAllocations(m)
        end if
    end loop
end
```

Figure 6.16. Finding the dataflow of allocations

is the minimum result of all recursively invoked analyzes. This result is returned and stored in the node. To prevent infinite recursion, all nodes are initialized to be accessed from `Integer.MAX_VALUE`. If the attribute is not equal to that when a node is about to be analyzed, the previously stored result is returned directly.

When analyzing method nodes (that represents return values), the analysis is modified as presented in Figure 6.18. As before, the method first checks if the node has been analyzed before by checking the value of the `accLevel` field. If not, the analysis continues. Since, these nodes represent returned values they have no send-to edges. When analyzing method nodes, the analysis progresses by analyzing all nodes accessed via return-to edges with the current level - 1 as input parameter, and via pass-to-edges with the current level as input. Note that if a returned value is passed to a method, it corresponds to passing the returned value to another method, i.e. a method at the same level as the current one.

All local nodes are treated as general nodes if the method that they belong to has been analyzed. However if it has not, the method is looked up in a table of safe methods. A safe method is a method that does not store or return its arguments. The result of calling a safe method is the same as the input, i.e. the object is not passed to any stack level above the current level. If the method is unknown, the worst case must be assumed. Thus -1 is returned which is equivalent to store the object in a global variable.

Finally, all objects that may be stored in fields or thrown as exceptions can not be analyzed with this design, therefore the analysis returns -1 when such case is detected. Returning -1 corresponds to saving a reference in a static variable and these objects should be handled by the runtime GC. How to handle fields is discussed in Section 6.6.

```
algorithm markDataFlow(Node this, int level)
    if this.accLevel ≠ Integer.MAX_VALUE then
        return accLevel
    end if

    int min ← this.level

    foreach n in this.sendTo do
        min ← min(min, markDataFlow(n, level))
    end loop
    foreach n in this.passTo do
        min ← min(min, markDataFlow(n, level+1))
    end loop

    this.accLevel ← min

    return min
end
```

Figure 6.17. Marking the data flow of general nodes

```
algorithm markDataFlow(MethodNode this, int level)
    if this.accLevel ≠ Integer.MAX_VALUE then
        return accLevel
    end if

    int min ← this.level

    foreach n in this.passTo do
        { Passing the return value to a sibling method }
        min ← min(min, markDataFlow(n, level))
    end loop
    foreach n in this.returnTo do
        min ← min(min, markDataFlow(n, level-1))
    end loop

    this.accLevel ← min

    return min
end
```

Figure 6.18. Marking the dataflow of return values

6.4 Code Generator Extensions

The escape analysis itself does not modify the generated code. Instead its output is communicated to the code generator that in turn uses this information to generate optimized code. Since the analysis is global, it needs information on all classes before it can produce any output. Thus, the analysis must be executed in a separate phase before the code is generated.

When source code is compiled into an executable format, the compiler can insert explicit free instructions according to the output of the analysis. There are several possibilities of how the data can be used.

An object can be allocated on the runtime stack when no reference to it escapes to any higher stack frame (or to global scope). This solution gives high performance since stack allocation is faster than heap allocation. Depending on the runtime support, it can or can not be used for objects with non-fixed size.

Another alternative that requires less work by the code generator is to associate sets of objects with each stack frame level. The sets contain the objects that should be reclaimed when the control returns from the stack level. The advantage of this solution is that it is easier to implement and it can also be used in interpreters such as the Java VM. The disadvantage is that the sets consume memory resources and it may be some what slower. Since the only operations on the sets are addition and iteration, the sets can be implemented as singly linked lists.

Objects may be kept alive longer than necessary they are allocated in a method that is called from different contexts. One invocation to a method that returns a newly allocated object might use the returned object temporarily, and another invocation to the same method may return the object to its calling method where it is further processed. Since both objects are allocated using the same instruction, they can not be reclaimed until control leaves the minimum stack level of the contexts that accesses any of the objects in the current design. Thus, the temporary object is kept alive for longer time than necessary. In the worst case, one object will be stored in a static variable which would prohibit all objects allocated by the associated allocation instruction to be handled by the static GC. An outline of how to solve this problem is described in Section 6.6.3.

Finally the code generator can eliminate all code that keeps track of objects that are handled by the static GC. The synchronization code can also be eliminated for all objects that are only accessed by one thread.

6.5 Limitations

This is the first incarnation of the static GC, and it has limitations. The following sections discuss these, and Section 6.6 outlines designs to overcome the ones that limit the system most.

6.5.1 Handling Fields

The current design does not handle objects that are stored in fields. One reason for this is that many field nodes may represent the same field in the graph. However, it is possible to merge all field nodes that may represent the same field in the same object. This information is in the graph, hence it is just a matter of extracting it. An outline of how this can be done is presented below.

In general, it is very hard to determine the life time of objects that are referred to by other objects. A safe assumption is to keep an object alive until all objects that may refer to it die, but that can waste memory since the object may have become unreachable long before that.

6.5.2 Large Systems Can not Be Analyzed

The execution time of the analysis phase is $O(AN_t)$ where A is the number of allocations and N_t is the number of nodes in the tree, and since $A < N_t$ it is $O(N_t^2)$. This becomes a problem since the size of the graph grows quickly when the graph is converted into a tree. The size of the resulting tree is $O(I^C)$ where I is the maximum number of call edges to a node and C is the maximum number of clones on any path from the root to a leaf (and clones occur whenever a method is called non-recursively from more than one other method.) Thus, the execution time is exponential in the number of clones on a path from the root to a leaf. To get around this, the conversion must be avoided. A design of a solution is presented in Section 6.6.2.

The current solution is to only analyze part of the system. As soon as data leaves the part of the system that is analyzed it is assumed that it is saved in global scope, and thus should be handled by the runtime GC. To be able to optimize the system, a list of safe methods can be input to the analyzer. These methods must not let any references that are passed to them escape to prevent from interfering with the analysis. Apart from the fact that manual techniques are tedious and error prone, there is another disadvantage of not analyzing the complete system. If a method that is not analyzed invokes a method that is, the return value may be used in a way that is not expected. Thus, to be sure of the correctness of the analysis, references must not escape from analyzed methods that are called from methods that have not been analyzed.

6.5.3 Objects Are Kept Alive

When using the technique where objects are stored in sets that are freed when methods return, objects are only reclaimed at the end of method invocations. This may cause objects that die to be kept alive longer than

necessary. This is often not a problem, but if, for example, an object is created in the main method while the application initializes, and then is not used anymore, it will still be kept alive until the end of the method. An inter-procedural def-use analysis would make it possible to reclaim many such objects immediately when they die. That is even better than dynamic GCs which can not free an object before it becomes unreachable.

Objects may also be kept alive because a method that allocates an object is called from different contexts. An outline of a solution of this problem is presented in Section 6.6.3.

6.5.4 Exceptions Are Not Handled

All objects that are thrown must be handled by the runtime GC using the current design. If exceptions are used in exceptional cases only, they should occur rarely and poses no major impact of the performance of the dynamic GC. Neither do exceptions cause problems when manually predicting their memory usage. Thus, this limitation does not restrict the applications that use the static GC in practice.

If exceptions must be handled statically, they can be stored in thread local variables, since there can only be one exception per thread in most cases. If exceptions can be wrapped in other exception, a stack of exception may be used. This work around does not work if references to the thrown exceptions are stored in the exception handlers.

It should be noted that handling exceptions statically can be done similar to handling return values. Thus, the problem is not hard, but it has low priority.

6.5.5 Finalizers Are Not Handled

The Java finalizers are methods that are used to free up resources that can not be freed by the GC. They are comparable to the destructors of C++. However, there is a major difference in that destructors are invoked when objects go out of scope or when they are explicitly deleted, and finalizers are invoked before the memory that is occupied by the object is reused. Another difference is that finalizers are invoked asynchronously in a separate thread. Since we can not tell when objects die, it is impossible to tell when the memory is reused. Another complication of finalizers is that a finalizer can make its object reachable again. Thus, objects can be reincarnated in the finalizer and then it shall not be freed until it becomes unreachable again. Even if an object is reincarnated, its finalizer should only be executed at most once.

A conservative approach is to add invocations to all finalizers in all objects that define them. The finalizers are executed in a separate thread, so

they can be inserted into the graph with a common parent method in a separate thread. To be sure that the finalizers do not cause any problems, all objects that are accessed by the finalizers should be handled by the runtime GC. Note, that since the `this` reference is passed to the finalizer, an object with a finalizer must be handled by the runtime garbage collector. Using this technique, finalizers can be used in applications that use the static GC.

Another possibility may be to handle finalizers as destructors if its corresponding object is handled by the static garbage collector. Since the static garbage collector makes the point where the object is reclaimed explicit, it can also invoke the finalizer that could be treated as any method. However, if the finalizer reincarnates the object, one has to be very careful.

6.6 Overcoming Limitations

6.6.1 Fields

The problem of handling fields originates from that multiple field nodes can represent the same actual field. The problem can be solved by adding edges from the allocation nodes to the fields that may be part of the object in a separate phase before the analysis. During the analysis, when a reference to an object is stored in a field, special edges from the stored object to the objects that may contain the field will indicate that one object may be stored in a field of other objects. The minimal stack depth from which an object is accessed from can then be calculated as the minimum stack depth according to the data flow of the object (which is stored in the method node) and the minimum stack depth of all objects that it may be stored in.

6.6.2 Large Systems

The major disadvantage of the technique presented above is its inability to handle large programs. The source of this restriction is that the call graph is converted into a call tree to make the analysis possible.

The conversion is necessary to associate a stack depth to each method invocation, and it is also used to separate calling contexts of methods that are called from more than one method. However, if we aim to calculate the stack depth of which an object can be reclaimed relative to the stack depth where it is allocated at, the stack depth of method invocations can be ignored. If we ignore fields at first, the only way an object can escape a method is by being returned. The relative stack depth is simply decreased by one for every time the reference is returned and increased by one every time it is passed as an argument. When all reachable nodes have been found, the minimum level is stored in the method node. Thus, the tree is

no longer necessary to track the relative stack depth from which an object is accessed if fields are not handled.

The approach used to handle fields, as described in Section 6.6.1, is to keep objects alive until all objects that may refer to them dies. As described, there are edges indicating that an object is stored in a field of another object. To be able to tell how long the referred object should be kept alive, we must be able to tell the relative stack depth distance between the allocations. This information can be found by following the data flow backwards from the point where the reference is stored to the allocation of the object that contains the field. Again, the relative stack depth is decreased every time a return-edge is traversed and incremented by one every time a pass-edge is traversed.

If the data flow passes a field node, all field nodes that may refer to the same field in the same object must be examined. To be able to find the relative distance between the allocations, the calls edges can be examined to find a common ancestor (there must be one or else the objects are allocated in different threads and can not be handled anyway.) If a common ancestor is found, the task to calculate the relative distance becomes trivial. This method can always be used if other approaches fail.

Hence, the relative stack depth distance of the allocations can always be found, and thus it is possible to find the minimum stack depth from which objects are referred from.

Using relative stack depths, one should be careful when using recursive methods. The presented analysis can only reclaim objects when the topmost invocation has finished its execution.

The WCET of the analysis is now $O(N_g^2)$, where N_g is the number of nodes in the graph, instead of $O(N_t^2)$, which is a remarkable improvement since the N_t grows exponentially compared to N_g.

To further improve the execution time of the analysis, dynamic programming can be used. By analyzing the methods of the graph in a bottom-up manor, and making shortcuts wherever possible, a sub-graph need only be analyzed ones. If shortcuts are made, it is important to record the number of stack frames that are skipped.

6.6.3 Calling Methods from Different Contexts

When a method that returns an object that it allocates is called from different contexts, the minimum stack depth from where the object is accessed can vary from context to context. The current design is conservative and reclaims the allocated object when leaving the minimum stack depth from which it is accessed in all contexts.

A better solution is to pass the minimum stack depth from which the object is accessed as a parameter to the method that allocates it. This information is available in the graph, so it can easily be added to the call. The

same approach can be used for objects stored in fields, by following the data flow of the objects containing the field.

6.7 Summary

This chapter introduces a static garbage collection technique that comes a long way in moving the garbage collection work from runtime to compile-time. The two major disadvantages is its execution time and its lack of handling objects that are stored in other object's fields. Solutions to both these problems are presented in Section 6.6. The technique has been implemented in the Jamaica VM [Sie02] and is presented in Chapter 7.

The static garbage collector will make it possible to run many Java applications without any runtime garbage collector. The only objects that can not be handled by the static garbage collector are objects that are part of the global state of the application and objects that are passed between threads. If these objects can be statically allocated, no runtime garbage collector is necessary. No previously published static garbage collection technique with reasonable execution time can eliminate the runtime garbage collector for such large class of applications. Unfortunately, the presented implementation does not have a reasonable execution time for large systems, but that has been fixed in the design presented in Section 6.6.2. However, this design has not yet been implemented.

— As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent finding mistakes in my own programs.

Maurice Wilkes

Chapter 7

Implementation

This chapter presents implementations of the real-time reference counting method and the static garbage collecting method. RTRC was first implemented as a set of CPP macros and a small library of functions. To use it, the developer has to make explicit calls to perform GC operations. RTRC was also implemented in a Java compiler that was built using the CoSy framework [ACE] and in the Jamaica JVM [Sie02]. Furthermore, the static garbage collector was implemented in the Jamaica builder that compiles Java to native executable code.

7.1 C implementation of RTRC

A first implementation of RTRC was developed using the CPP macro processor and a small library of C functions. Developers can easily maintain full control of the emitted GC operations, since all RTRC operations are explicitly invoked. This is also a disadvantage since mistakes can lead to a memory leak or prematurely reclaimed objects.

The interface to the implementation consists of functions and macros. The functions are used for: *initialization*, *allocation*, and for *preparing blocks* for allocation by high priority threads. The increment and decrement operations are implemented as macros for performance. Macros for assignment, function call, and function return have also been implemented to increase usability. An example using the C version of RTRC is presented in Figure 7.1.

The implementation is configured using macros. One macro enables the reference counter. If this is not set, reference counting completely disabled. Another macro enables statistics counters that count the number of invocations of the increment and decrement operations, as well as the number of actual increments and decrements. The statistics can be used to

```
flow_t *flow_create(double flow, double a,
                    tank_t *from, tank_t *to)
{
  flow_t *f;

  /* Increment the parameters */
  RC_INCR(from);
  RC_INCR(to);

  /* Allocate an object */
  f = flow_alloc();

  /* Assigning primitive members */
  FLOW__FLOW(f) = flow;
  FLOW__AREA(f) = a;

  /* Assigning references */
  RC_ASSIGN(FLOW__FROM(f), from);
  RC_ASSIGN(FLOW__TO(f), to);

  FLOW__CALCULATE(f) = calculate_pipe;

 cleanup:
  /* Decrementing locals and parameters except */
  /*  the value that is returned               */
  RC_DECR(from);
  RC_DECR(to);

  return f;
}
```

Figure 7.1. A function using the C version of RTRC

help verifying the correctness of the use of the reference counter. The size of the blocks and the number of blocks on the heap can also be customized.

This implementation does not automatically split objects, since it can not be done transparently without modifying the compiler. The implementation is presented in Appendix A and benchmarks are presented in Section 8.2.

7.2 CoSy Implementation

The real-time reference counting technique described in Chapter 4 has also been implemented in the JOSES[1] Java-to-native compiler (JOC) [RBLP00, Vee01].

[1] ESPRIT LTR project #28198 "JAVA and CoSy Technology for Embedded Systems"

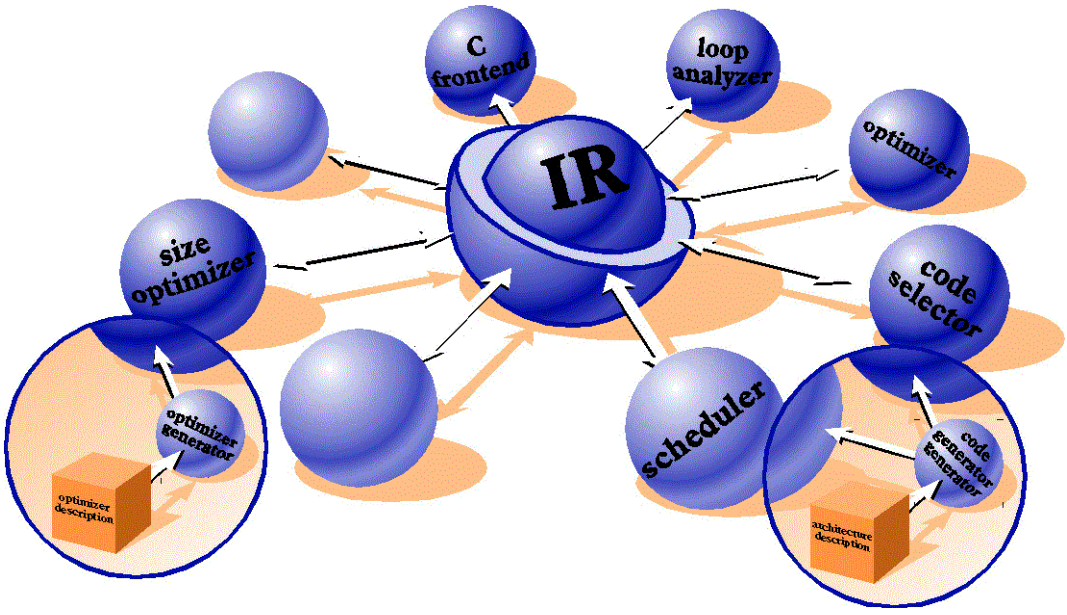


Figure 7.2. CoSy

7.2.1 CoSy

The CoSy compiler framework from ACE Associated Compiler Experts was used to develop JOC, the JOSES compiler. CoSy was first developed within the Compare ESPRIT project [AvSM93]. A CoSy compiler is built using modules called *engines*. These engines communicate via a shared memory called common data pool (CDP). To access the CDP, the engines must define the structure of the data. These definitions are used to generate the data manipulation and control package (DMCP) library, which is called by the engines to access data in the CDP. The structure of the data in the CDP is defined using the full structure definition language (fSDL). A CoSy compiler is constructed by combining engines using the engine description language (EDL).

CoSy Engines

Engines can be programmed in any language that can interface with C, i.e. most modern languages. The engines in a compiler usually include frontends, optimizers, lowerers, and backends. The frontend reads the source language and generates the intermediate representation (IR), which is used to represent the input to the complier. The IR is then passed to optimizers and lowerers to be transformed. A lowerer is an engine that transforms the IR to a simpler form to make backends (and possibly optimizers) easier to implement. A lowerer can, for example, transform switch statements into if statements. The IR is finally passed to the backend that generates the output of the compiler.

Engines can also be used to analyze the program and annotate the IR to simplify the work of other engines. An example is the loop analyzer, which finds loop constructs and extracts initialization, condition, advancement, and body information.

The interface of an engine is defined in EDL. Since different engines have different requirements on the IR, engines can specify their own view of it. The DMCP library only allows access to the IR according to the specified view of the engine.

A tool, called BEG (back-end generator) [ESL89], has been developed to ease development of backends. A BEG engine scans the IR for patterns that it can translate into the destination language. If several transformations are possible, the cheapest one is selected according to the cost associated with every transformation in the specification of the backend.

The Engine Description Language — EDL

The engines of a CoSy compiler are glued together using the Engine Description Language (EDL). EDL supports six possibilities to combine engines into a compiler or into a composite engine.

pipeline The output of one engine is passed as input to the next.

data-parallel The IR can split into disjoint subsets that can be processed in parallel, e.g. one instance of an engine per procedure.

fork Several independent engines work in parallel.

loop As in a pipeline the data is passed from one engine to the next. However in a loop, a status engine determines whether the loop should end or continue. If the loop continues, the data is passed from the last engine to the first, otherwise data is passed to the engine following the loop.

speculative Several engines modify the IR graph. The result is passed to a selector engine which determines which result should be used and the results of the other engines are discarded. The output is that of the selected engine.

optimistic First one engine is used to generate several potential solutions for a specific problem. A status engine determines when enough work has been done and stops the first one. A third engine selects which results (IR) are to be considered for further processing. The IR that passes the third engine is passed on to an instance each of a fourth engine. All instances of the fourth engine work in parallel. Results that are rejected by the third engine are discarded. Each instance of the fourth engine continues to processes the IR and finally a fifth engine chooses which IR is to be used and discards the rest. The chosen IR is passed on to the next engine.

term	::=	icon | rcon | scon | const
		| op { fields } | [list]
fields	::=	$\text{field}_1 = \text{term}_1, \ldots, \text{field}_n = \text{term}_n$
list	::=	$\text{term}_1, \ldots, \text{term}_n$

Figure 7.3. BAR syntax

The Intermediate Representation – CCMIR

The CoSy system does not force engines to use a particular IR, but since supplied engines use the Common COMPARE Medium Intermediate Representation (CCMIR), the use of this IR is recommended. Engines can easily extend the IR without interfering with existing engines and the extensions are hidden from engines that do not use them. CCMIR is defined in fSDL as is all data that is communicated between engines.

CCMIR is defined to be language-independent. Currently most imperative and object-oriented constructs are supported. The object-oriented extension was designed within the JOSES project. Other extensions, e.g. a DSP (Digital Signal Processor) extension, already exist.

BAR — an Interface to CoSy

To simplify frontend development we have designed an ASCII representation of CCMIR called BAR. To convert BAR into CCMIR a BAR frontend for CoSy has been developed. The frontend can be combined with any CCMIR engine to create an industrial strength optimizing BAR compiler. So far prototype compilers for Pentium II and Sparc have been developed.

BAR was originally developed to connect frontends written in the Relational Meta-Language (RML) [Pet95], a natural-semantics like specification language, with the CoSy system. BAR makes it possible to use virtually all frontend construction tools available. The only requirement is that ASCII files can be produced. BAR also makes it easier to debug the output of frontends.

A BAR file contains a sequence of terms. A term can be an operator, a constant, or a list of terms. Operators may have attributes, which can be set to values represented by terms. Some attributes are mandatory, for others default values can be calculated, e.g. for alignment of types. The complete syntax is shown in Figure 7.3.

An example is presented in Figure 7.4 and Figure 7.5 where the C version of “Hello world” has been translated into BAR.

```
int printf(char *, ...);

int main() {
    printf("Hello World!\n");

    return 0;
}
```

Figure 7.4. Hello World program in C

```
Integer {Name = "int", Size = 32}
Integer {
  Name = "char", Size = 8, Signed = FALSE
}
Pointer {Name = "char*", RefType = "char"}
ProcType {
  Name = "printf", ReturnType = "int",
  Params = [Parameter{Type = "char*"}],
  MoreArgs = TRUE
}

ProcGlobal {
  Linkage = ImportLinkage, Type = "printf",
  Name = "printf"
}

ProcType {
  Name = "main", ReturnType = "int", Params = []
}

ProcGlobal{
  Linkage = ExportLinkage,
  Name = "main",
  Body = [
    Call {
      Proc = "printf",
      Params = [ CStringConst {Value = "Hello World!\n"} ]
    },
    Return {Value = IntConst {Value = 0}}
  ]
}
```

Figure 7.5. Hello World program in BAR

7.2.2 The Reference Counter Engine

The `refcount` engine introduces operations to maintain reference counts in the IR. This is done in one or two passes through the code. The first pass introduces calls to the functions `ckf_incrc()` and `ckf_decrc()` to perform the actual reference count updates and the optional second pass inlines these operations.

The engine works on the object-oriented layer of CCMIR, but it could easily be adapted to work older version of CCMIR to support non-object-oriented languages. However, the objects which should be handled by the reference counter must then be marked somehow. In the current implementation, all object instances are reference counted.

The reference counter requires at most three temporary variables per function: one to store the return value during cleanup (if the function returns a value), and two that are needed when emitting assignments and function calls.

When `refcount` starts, the `Object`-type is located, i.e. the type that all other objects directly or indirectly inherits from, and the union type needed to store the reference count and next pointer is created. Then one dereference function is created for each class. The dereference function decrements the reference counts of all children of an object. This implementation decrements all children and can not take one block at the time, which is required in a full implementation.

Then all functions are processed. If a function returns a value, a temporary variable is created to temporarily store the value. A new basic-block is created to clean up the activation record when the function is to return. Cleaning up is done by decrementing the reference counts of the objects referred to by all parameters and local variables except for the return value.

In the next step all statements in the function are processed. The first statement in all functions is the begin statement. Following that all local variables which refer to objects are initialized to null and all reference counters of objects referred by the actual arguments are increased.

The assignment statements referring to objects are the most complicated. The right-hand-side of the assignment is stored in a temporary variable and incremented. The address of the left-hand-side is stored in another temporary and decremented. The temporaries are used since the expression could have side-effects and should then not be evaluated more then once. Finally the value of the right-hand-side is stored in the address of the left-hand-side.

Return statements are changed into an assignment to the return value temporary variable (if the function returns a value) and a goto to the clean-up block.

Each function call which store a return value that holds a reference to an object is updated to first save the reference that will be replaced by the

return value in a temporary variable. After the call, the reference count of the object referred to by the temporary variable is decremented. The reference count can not be decremented before the call, since that could cause the object to be prematurely reclaimed. An alternate approach to handling function calls is to increment all actual arguments referring to objects before the call. That would eliminate the need for saving the reference that is updated by the return value, but it would increase the code size (given that functions are called one or more times.)

When all functions have been processed the reference count union, i.e. the union that contains the reference counter or the next pointer as described in Chapter 4, is added to the `Object`-type, and the reference count operations are optionally inlined.

7.2.3 The Record Splitter Engine

The record splitter works on CCMIR without the object-oriented features. Thus, all object instances have been transformed into records. To help finding the records to split, the lowerer that removes the object-oriented constructs was updated to mark the object instance records. However, some objects can not be divided since they are used as global or local variables or as parameters. This is only an issue for objects that represent classes in JOC since they are stored as global variables. To verify this, the IR is traversed and object records that are used as global or local variables or parameters are added to a set of instances that should not be split.

Next, the engine iterates over the list of all types to find object records. The fields of records which are in the set described above are prepended with a next field (a void pointer), so they get the same prefix fields as other objects (next pointer, type information and reference counter.) All other object records are divided into blocks.

While a record is split, a new block type is created and a next field is added to it. The original type is associated with a list of the resulting block types by storing it in a map. The fields of the object record are added to the new block type until the size of the block type would exceed the block size. New block types are created as needed. The block number and offset of each field is stored to make it possible to find the field using reflection at run-time. The number of blocks needed by an object is also stored to be used when allocating objects at run-time. All information needed at runtime is stored in the initialization expressions of the type information.

In the next phase, all member accesses are modified to use the next fields if necessary. This is done by checking the type of the object record and using the map to find the list of block types. These blocks are then searched for the field. For every block passed, a next reference is generated.

All class record initialization are then prepended with a null pointer for the next field. And finally, all instances of pointers to replaced object records are replaced by pointers to the first block type in the associated list.

Currently arrays are not split, but this can easily be implemented by modifying array subscript implementations, and adapting the array allocation function.

7.2.4 The Runtime System

The design of the runtime system is crucial to the performance of compiled code. Several issues have to be handled, e.g. virtual method calls, exceptions handling, and interfacing to native code.

Virtual Methods

Calling a virtual method in a language that only supports single inheritance is cheap and trivial. All classes keep a table, called virtual table, of virtual methods. All virtual methods in a class get a unique index in the table. If a method is inherited from a superclass, its position in the table is simply set to refer to the superclass' method in the sub-class' virtual table. An object can be cast to a superclass, since the virtual tables of all superclasses are prefixes to the sub-class' virtual table. This is where the problem with multiple inheritance occurs. A superclass' virtual table is not automatically a prefix of the sub-class' virtual table, since a class can have multiple superclasses. Thus, when multiple inheritance should be supported, that simple schema does not work.

The Java interfaces cause the same problem as multiple inheritance in regards to calling virtual methods (except for possible ambiguity.) In Java the problem arises when a reference is cast to an interface, the interface's virtual table can't be used (since it is not a prefix of the virtual table of the object's class.) Standard implementations [Str94] make references point into objects (rather then to its beginning) to indicate where to find the virtual table of the interface. The virtual tables of the example in Figure 7.6 is presented in Figure 7.7. To support this schema, the object headers must include pointers to all virtual tables whose virtual tables are not prefixes of the object's class' virtual table. After the virtual table pointers, offsets to the beginning of the object are kept to be able to find the start of the object. The offsets are needed to find the instance variables and to be able to convert a reference to other classes and interfaces. This schema give an overhead in all objects, and requires three memory dereferences at every call, and one dereference for every instance variable access (to find the offset.) The source code in Figure 7.8 shows how to call methods and access members using this method.

The JOC implementation decrease the per object memory overhead and the number of memory dereferences by increasing the per class memory overhead. Every class has an interface table which holds references to virtual tables of all implemented interfaces. Every interface gets a globally

```
interface I1 {
    void f1();
}

interface I2 {
    void f2();
}

class A implements I1 {
    public void g() {}
    public void f1() {}
}

class B extends A implements I2 {
    public void f() {}
    public void f2() {}
}
```

Figure 7.6. Java example for virtual tables

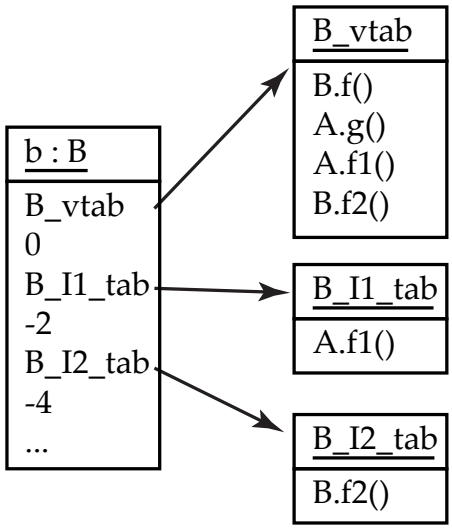


Figure 7.7. Virtual tables using the standard implementation

```
// Calling a method (2 dereferences)
obj->vtab[METHOD_ID]()

// Accessing in instance variable (2 derefrerences)
(obj + obj->offset)->member
```

Figure 7.8. Calling methods and accessing members using the standard C++ implementation

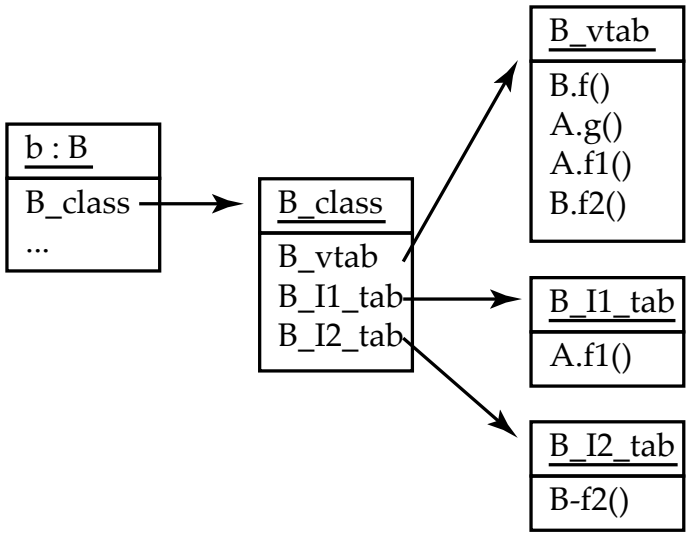


Figure 7.9. Virtual tables using JOC

```
// Calling a method (3 dereferences)
obj->class.itab[INTERFACE_ID][METHOD_ID]()

// Accessing a member (1 dereference)
obj->member
```

Figure 7.10. Calling virtual methods and accessing members in JOC

unique index. The length of the interface table is the largest index of the implemented interface plus one. The first position in the interface table is used to store a pointer to the class' own virtual table. The interface tables of the example in Figure 7.6 is presented in Figure 7.9, and source code that calls a method and accesses a field is presented in Figure 7.10.

By making all interface tables be of equal length (the number of interfaces) the class' virtual table can be located at a fixed position in the class record. Thus, when the type of the reference used to make the call is a class, one dereference can be omitted. The overhead is presented in Table 7.1

Exception Handling

There are two main approaches to handle exceptions. Either a direct jump is done to the nearest exception handler, e.g. using the C `longjump()` function, or the stack is wound up by returning from all functions until an exception handler is found. A direct jump is generally faster when an actual exception is thrown, but it slows down execution when entering a try/catch block since registers and such data needs to be saved. The main disadvantage of stack unwinding is that when a function that can throw

Operation	JOC	Std. implementation
Virtual calls	2 dereferences	2 dereferences
Interface calls	3 dereferences	2 dereferences
Attribute access	1 dereference	2 dereferences
Object size	1 word	2 words per superclass
Class record size	1 word per interface	none

Table 7.1. The overhead of multiple inheritance in JOC and using the standard (C++) implementation

an exception returns, it must be checked if an exception has been thrown or not. However, certain run-time systems require some cleanup when returning from functions, e.g. decreasing reference counts and calling destructors. To be able to use the `longjump` solution, these cleanup blocks must be registered and called when passing them during a `longjump`. Since JOC uses reference counting, which requires the references of all local variables and arguments to be released, the `longjump` solution would most certainly be slower than using stack unwinding. Thus, JOC uses stack unwinding.

Since exceptions are local to threads in Java, the exception handler must be aware of threads. Every function generated by JOC takes an environment as the first argument. This environment is unique for each thread. One field of the environment is used to store a possible exception. Throwing an exception is a matter of setting the exceptions field and jumping to the nearest exceptions handler in the function. If there are no exception handlers in the function, one is created. If a thrown exception is not handled by the handler, the handler jumps to the next following handler. If the handler is the last in a function and the exception is not handled by it, the clean up code is invoked and the function returns. When a function that can throw exceptions returns, the environment has to be checked. If an exception has been thrown, the handler jumps to the nearest enclosing exception handler. This goes on until a handler that handles the exception is found. If a handler has a finally block it is always invoked regardless if the exception is caught or not.

Native Interface

The Java Native Interface (JNI) is used to combine Java code with code written in other languages such as C. This interface sets up an environment for the native code to execute in. The environment wraps access to data in the virtual machine and maintains the runtime information. Among the functionality provided in the JNI are mechanisms for local and global references. A global reference is a reference that has to be released like any

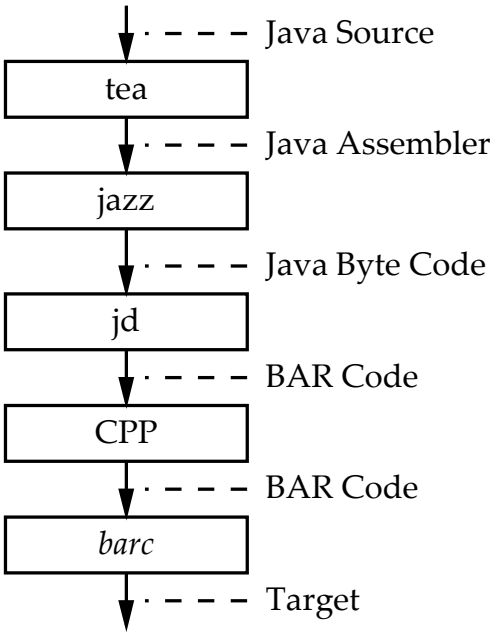


Figure 7.11. The structure of JoC. The barc compiler is presented in Figure 7.12

.

reference created by the Java code. Local references, on the other hand, are automatically released when leaving the native code. This is implemented by storing a linked list of local references in the environment. All native calls from the Java code are wrapped with an initializer that initializes the environment. When the native call returns, the environment is examined to see if an exception has been thrown, and if there are no other active native methods on the stack, the local references are released.

7.2.5 The Compiler

The full compiler, presented in Figure 7.11, which compiles Java source to target code (currently C or Pentium assembler) includes several modules. First the so called `tea` compiler translates Java source into "Java assembler". This compiler was originally developed in two master projects at PELAB [Com97, Hol00]. The `tea` compiler was automatically generated from a Java semantics specification in RML [Pet95], a natural semantic specification language developed at PELAB.

The Java assembler output from `tea` is translated into Java byte code using the `jazz` assembler also developed at PELAB. This stage was introduced to simplify offset calculations.

The byte code is translated into BAR code to interface to the CoSy compiler. The translation is performed by the `jd` compiler, which is also im-

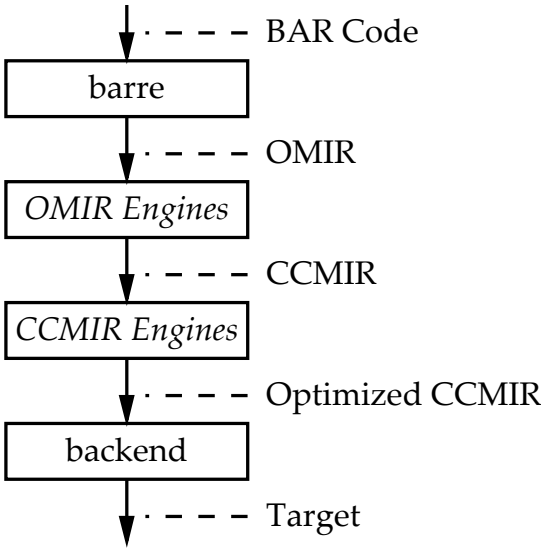


Figure 7.12. The structure of `barc`, the BAR to CCMIR translator. The OMIR engines are presented in Figure 7.13 and the CCMIR engines are presented in Figure 7.14.

plemented in RML. The implementation was done in three PELAB master projects [Ben98, Roo00, Kar00] and was later almost completely rewritten to emit the object-oriented version of CCMIR. Since the byte code is stored in one file per class, this was also chosen as the compile unit for `jd`. Thus, the output is one BAR file per class.

Before the BAR code is passed to the CoSy compiler, all BAR class files are concatenated and passed through the C macro processor (CPP). CPP is only used for inclusion of standard types and class specifications (also generated by `jd`). By concatenating all class files, a single huge BAR file is generated even for small Java programs. This was chosen for simplicity, since inter-class optimizations were planned for JOC.

Finally the BAR code is passed to the `barc` compiler which is implemented in CoSy. The output from `barc` is either C code or Pentium assembler. This stage is further described below.

The BAR Compiler — barc

The first stage of `barc`, as presented in Figure 7.12 is the BAR REader engine, `barre`. This engine reads the file containing the complete Java application and translates it into CCMIR. The IR is then passed to engines that optimize it before it is passed to the engines that remove the object oriented features of the IR. Next follows engines that further optimize the code on the old CCMIR level. Finally the backend generates the target code.

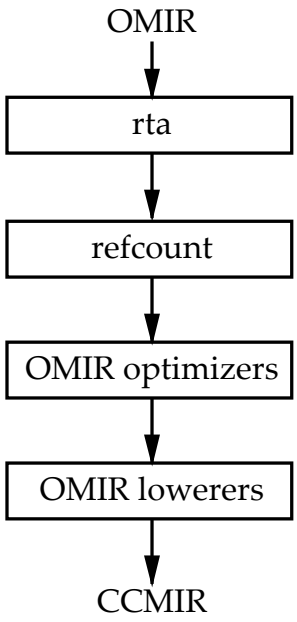


Figure 7.13. The OMIR engines of `barc`.

The OMIR Engines

While the object-oriented version of CCMIR was developed, the object-oriented subset of CCMIR was named OMIR. At that stage we divided the engines into engines that were designed to handle OMIR and the older engines that did not expect any OMIR. This section describes the engines that were designed to work on OMIR. The structure of the OMIR engines is presented in Figure 7.13.

The `rta` engine optimizes the code and decreases the load on the other engines by using rapid type analysis [Bac97] to remove classes and methods that are never used by the application. This greatly reduces the compilation time. After that, the `refcount` engine is invoked to add reference counting operations to the code (see Section 7.2.2.)

The IR is then passed to the OMIR optimizers. However, no such engines were available when the compiler was implemented. Therefore, the IR is just passed on to the OMIR lowering engines. The OMIR is lowered in three steps. The object-oriented nodes, such as classes, instances, member access, and dynamic method calls, are first translated into lower level CCMIR. This stage also creates the run-time type information. Next, the interface to native code (normally written in C or C++) is wrapped to implement the Java Native Interface [Lia99] standard. Finally the exceptions are translated into lower level CCMIR. In the evaluation compiler, the exceptions are simply removed. The resulting IR no longer contains OMIR.

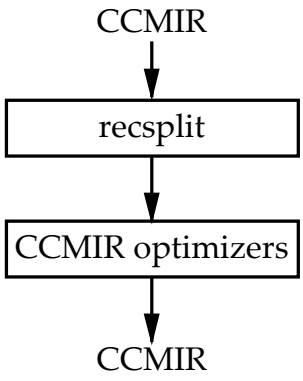


Figure 7.14. The CCMIR engines of barc

7.2.6 The CCMIR Engines

The CCMIR engines, i.e. the engines that work on non-object-oriented CCMR, are presented in Figure 7.14. The IR is first passed to the `recsplit` engine, which is described in Section 7.2.3. After that the CCMIR optimizations engines are invoked. The optimizations include constant propagation and dead code removal. The IR is finally passed to the backend that currently generates C code.

7.2.7 Future Work

The JOSES compiler is a prototype and is currently not useful for "real" applications. The compiler executes too slow and the generated code is not good enough. Some future improvements are discussed below.

Temporary Reference Variables

The JOC frontend generates many temporary variables. This is partially caused by the design of CCMIR, but can still be greatly improved. The problems of generating many temporaries are that locality is destroyed, memory usage increases, execution time increases since temporary references are also reference counted, and all objects are not reclaimed immediately since temporary variables still reference them when the original reference is updated.

There are two approaches to decrease the number of temporary variables. One is to rewrite the frontend (`jd`), and the other is to write an engine that reuses temporaries and removes redundant ones. To get the best improvement both approaches should be combined. Such an engine can be implemented using a def-use analysis to find out if a temporary variable could be reused, and a data flow analysis to find redundant ones.

To be able to reclaim objects as soon as possible, temporaries should also be set to `null` after their last use.

Compilation Time

The compilation time of non-trivial Java programs using JOC is far too long (5 – 20 minutes far a small benchmark). The reason is that all code of the application is compiled into one big unit. To be able to perform inter-class optimizations the classes from the run-time library are also included. A "Hello world!" application requires over 100 classes and about 150 MB of memory to compile. These figures are from using the rapid type analysis engine which removes much of the redundant classes before passing the IR on.

First, separate compilation should be allowed. To support inter-class analysis, a database must be kept between compiler invocations. Secondly, the compiler should be lazier and only compile what is necessary. A static call tree could be maintained during compilation and only the methods which could be called should be compiled. These approaches would require a massive rewrite of the compiler and will, because of the effort required, not be implemented in the JOSES compiler.

7.3 The Jamaica VM

The Jamaica system, developed at aicas GmbH, is a hard real-time Java implementation with an interpreter and a compiler. Interpreted and compiled code can be mixed to optimize speed and memory usage. There are actually two interpreters, one that executes class-files directly, and one that uses a more compact class format. Using the latter, an executable file is generated. This program embeds the interpreter and the classes. Currently, Jamaica generates C code, but a more flexible back-end is under development. The new back-end will be able to generate native code for several platforms.

The Jamaica runtime system divides the heap into blocks that are collected using a mark-and-sweep garbage collector. Small objects are built by linking blocks into a list, and larger objects use a tree structure. If it is possible, larger trees are contiguously allocated to improve average execution time.

7.3.1 Real-Time Reference Counting

At first, it was planned to integrate the RT-reference counter into the interpreter. However, implementing a reference counter directly in the JVM is very inefficient because of the nature of the Java byte-code. The easiest approach is to rewrite the code that pushes and pops references to and from

```
Method void main(java.lang.String[])
  ; RedundantRC obj = new RedundantRC();
  ; Allocate an object and push it on the stack (rc = 1)
  00 new #2 <Class RedundantRC>
  ; Duplicate the reference (rc = 2)
  03 dup
  ; Invoke the constructor on the top element
  ;    of the stack (rc = 1)
  04 invokespecial #3 <Method RedundantRC()>
  ; Store the reference in local variable
  07 astore_1

  ; int i = obj.i;
  ; Push it back on the stack (rc = 2)
  08 aload_1
  ; Get the attribute and put it on the stack (rc = 1)
  09 getfield #4 <Field int i>
  ; Store the attribute in a local variable
  12 istore_2

  ; When returning the local variable is cleared (rc = 0)
  13 return
```

Figure 7.15. Redundant reference count updates in the JVM

the stack and the code that stores references in local variables and objects. However, this would introduce many more reference count updates than necessary.

Many reference count updates can be eliminated by introducing them at byte-code level, e.g. a reference that is moved from the stack to a local variable need no update. This is the best one can do without analyzing the byte-code first. Unfortunately this approach leaves many redundant reference count updates in the code. The byte-code in Figure 7.15 shows an example of reading an attribute of an object. Reading data is an operation that does not require any reference count updates, but the byte-code needs to copy a reference to the object containing the attribute to be stored on the stack. Since the reference counter can not tell how this reference will be used, it must be counted. The example requires five updates (not counting the initialization). If the reference count updates are introduced before the temporary variables (the stack) is introduced, only one update is necessary. Although it is complicated, this information can also be extracted from the byte-code.

Still, the reference counter was implemented in the interpreter as a proof of concept that it can operate with the mark-and-sweep garbage collector. The system runs, but no efforts have been made to benchmark it,

since it is too inefficient. Thus, it does not give any interesting indication about the efficiency of the technique.

The conclusion of this implementation is that the reference counter either should be implemented in the compiler or the byte-code need to be analyzed before it is executed for this technique to be competitive. A possible approach to perform the optimization is presented below.

Barth's Peep Hole Analysis

An approach to decrease the number of reference count updates has been proposed by Barth [Bar77]. By doing a simple peep hole analysis it is possible to find increment operations followed by decrement operations on the same object or vice versa. These updates can be discarded with no semantic change to the program. This simple optimization can be implemented by traversing the byte-code of a method and insert reference count updates. This code can then be optimized to eliminate reference count updates according to Barth's technique. These two steps can also be integrated to increase the performance of the optimizer, which is very important since it is invoked during runtime.

7.3.2 Static Garbage Collection

The extended escape analysis as described in Chapter 6 has been implemented. Since the modifications to the code generator have not yet been implemented, the escape analysis is not run in a separate phase. Instead its execution is interleaved with the code generation to eliminate a separate phase. The analysis has been fully implemented except for the support of multiple threads.

Generating the Graph

The only part of the analysis that is compiler dependent is the generation of the data and call graph as described in Section 6.3.1. This was mainly implemented by adding methods to the existing classes that represent the IR of Jamaica. All intermediate commands in the IR that cause any change to the graph implements the `buildDFG()` method to represent the data flow of the command. To be able to generate the graph, it is necessary to know the sources of the references. This information is found using the `dataSource()` method that is implemented in all commands that can produce a reference. The nodes of the graph are created using the `createDFGNode()` method that is implemented in all commands that may be the sources of references. Finally, the call graph information was added by the `createCallGraph()` method that adds the call edges. With a few exceptions, this was the only changes to the existing code that was

needed to implement the analysis. The commands that affect the data flow and the changes that they make to the graph are listed below.

New, NewAArray, NewArray, and NewMultiArray These intermediate commands allocate new objects and store them in a variable. A node for the allocation and a send-to-edge to the variable is created.

WriteVar This intermediate command stores a value in a variable. If the type of the value is a reference type, a send-to-edge is created from the source of the value to the variable.

GetData The `GetData` intermediate command reads a field of an object. It does not directly generate any nodes or edges. However, when it is used as the source of any intermediate command that creates a data flow, and no node in the graph represents the field already, it creates such a node and connects it to its object by adding a field-edge from the object to the field.

SetData The `SetData` intermediate command stores a value in a field. Three nodes are created if they do not exist: the destination object that contains the field, the field and the source (of the right hand side.) These are connected by adding a fields-edge from the destination object to the field, and a send-to-edge is added from the source object to the field.

ArrayStore The `ArrayStore` intermediate command represents a value that is stored in an array. If the value is a reference, a send-to-edge is added from the source of the value to the array.

GetClass The `GetClass` intermediate command represents an intermediate command that retrieves the class of an object. This would require a send-to edge from all classes to the destination node. However, since all classes end up on the heap (since they are global) they can be handled by a common dummy class node. A send-to-edge from the dummy node to the destination node is created.

TermReturn The `TermReturn` intermediate command is used to represent a value being returned from a method. If the value is a reference, a send-to-edge from the source of the value to the node that represents the method is created.

TermThrow The `TermThrow` intermediate command throws an exception. Since exceptions are not handled, a send-to-edge from the thrown object to a node that represents the `TermThrow` intermediate command is created.

Invoke The `Invoke` intermediate command generates the most complicated data flow. Since dynamic dispatching is used, one can seldom decide exactly which method is being called at compile-time. If multiple targets are possible, data flow has to be created for all possibilities. First pass-to-edges are added from the parameters that are of reference types to the corresponding formal arguments of the called method. If the nodes for the formal arguments have not yet been created these are created. If the return value is of a reference type, a return-edge is created from the method node to the variable that receives the return value.

The `Invoke` intermediate command is also used to generate the call graph of the system. If the called method is static, a calls-edge is added to the called method. Otherwise, a calls-edge is added to every possible target method, i.e. the method that is called and all methods that override it.

7.3.3 Debug Output

To debug the output of the analysis, the complete graph is output using the graph description language (GDL) of aiSee (previously called VCG.) The aiSee tool is developed by AbsInt (see `http://www.absint.com`.) The graph can be examined using the aiSee tool. This can be used to understand why some allocations can not be handled statically, and possible to improve it to make it possible handle more objects at compile time.

7.4 Summary

RTRC has been implemented as a C library and in a full Java compiler. The C implementation makes it possible to test the technique with full control of the emitted source code. It also makes it possible to test the impact of optimizations by manually making changes to the code. This way, the optimizations need not be implemented to be tested.

The RTRC implementation in JOC is useful to test the technique on larger systems, which are hard to implement correctly using the C implementation. Unfortunately, the JOC compiler is itself a prototype, which among other things results in that some compilations fail, some generated code fails, huge CPU and memory requirements, and in excessive generation of temporary variables. Since all temporary variables are reference

counted, the performance of this implementation of RTRC is low. However, it shows that the technique works, and it can be used as the base of other implementations.

The static garbage collector has been implemented in the Jamaica VM. Currently, the code generator has not been extended to make use of the results, but it can be used to estimate the memory requirements of systems and as a tool to find potential memory leaks in applications. The output of the analysis can be viewed as an annotated combined data flow and call graph.

— I love deadlines. I like the whooshing sound they make as they fly by.

Douglas Adams

Chapter 8

Benchmarks

This chapter presents benchmarks of the implementations that have been presented in Chapter 7. Benchmarking is a hard problem by itself, and benchmarking a garbage collector is even harder. Therefore, we start by discussing what to measure when benchmarking a garbage collector.

8.1 Benchmarking a Garbage Collector

Benchmarking is used to measure the performance of a system, e.g. a hardware platform, a compiler, or a ray tracer. It is common to test hardware and compilers using some standard set of benchmarks applications. These applications typically measure the execution time of some computations. When writing the benchmark applications one has to be careful not to make it possible to optimize away the computations, e.g. by constant folding or dead code elimination. Constant folding can be avoided by including an unknown in the computation, e.g. a random number, and the result can be output on the console to avoid some dead code removal.

Still, many computations can be optimized away, and that makes comparisons difficult. Some examples of such optimizations are presented below. In one test, the system performed about 20 divisions per clock cycle, which is quite impressive. It is also important to think about how the system is going to be used when reading the benchmark results, e.g. if you buy a processor that is 100 % faster than your old one, the system will not boot on half the time, and if a garbage collector performs 100 % faster in one test, it does not mean that it will in your system.

Thus, benchmarking in general is a hard problem, and when including a garbage collector it becomes even more complicated. As presented in Chapter 4, many garbage collectors perform differently when the amount of used memory varies (compared to the available memory). Therefore, the heap size becomes very important. If you have a large enough heap you

may be able to disable the GC completely. However, the execution time of RTRC does not depend on the heap size, i.e. measuring becomes simpler. But still, we do not know what to measure.

The execution time of an application depends much on the performance of the rest of the system, i.e. a plain timing would not give any interesting results. We decided to compare our system to a system that does not reclaim memory at all. This is not fair, since even manual memory management takes time, both for reclaiming the memory and sometimes also for housekeeping to tell when objects can be reclaimed. However, it is as close as we can get if we do not want to rewrite the benchmark applications to include manual memory management, and even that is not fair since we could make a poor job that slows down the application more than necessary.

Preferably, we should compare RTRC with other techniques, but that is even harder. To compare garbage collectors they should be implemented in the same system and the same effort should be made to optimize them all. The benchmarks presented in this chapter are benchmarks of a prototype GC in a prototype compiler with a prototype runtime system. To compare these results to industrial strength systems may not give a correct view of the potential of RTRC. To complicate things further, RTRC is a real-time garbage collector, and that makes comparisons against non-real-time garbage collectors even harder and maybe not very interesting since a non-real-time garbage collector can postpone all GC work, and if the heap is not exhausted the GC does not have to perform any work at all, while RTRC continuously performs the GC work.

Still it may be interesting to see approximately how other widely used garbage collection techniques perform. In a benchmark of a generational collector [BJMM02], the time spent in GC and the total execution time was measured. The size of the heap was compared to a "tight heap", which should be close to the size of the live memory plus the memory used for sub-heaps that are not in use, and the execution time was compared to the shortest measured execution time. For a tight heap the time spent in GC varies from 12 % – 35 % and the execution time increased 13 % – 40 % compared to the best run. When using a heap that is about 30 % larger than a tight heap the time spent in GC was 1 % – 20 % with an average around 10 % and the execution time increased with 3 % – 11 % compared to the best run. Note that the memory consumption is compared to a "tight heap", which is not necessarily the same as the maximum amount of live memory. And the execution time is compared to the best run, which also included garbage collection work.

Since RTRC is designed to work with extremely tight heaps, the results should be compared to those, and it is also important to keep in mind that the generational GC is no real-time GC and that the measures below are of a non-optimized RTRC, i.e. no object ownership and no static GC is used.

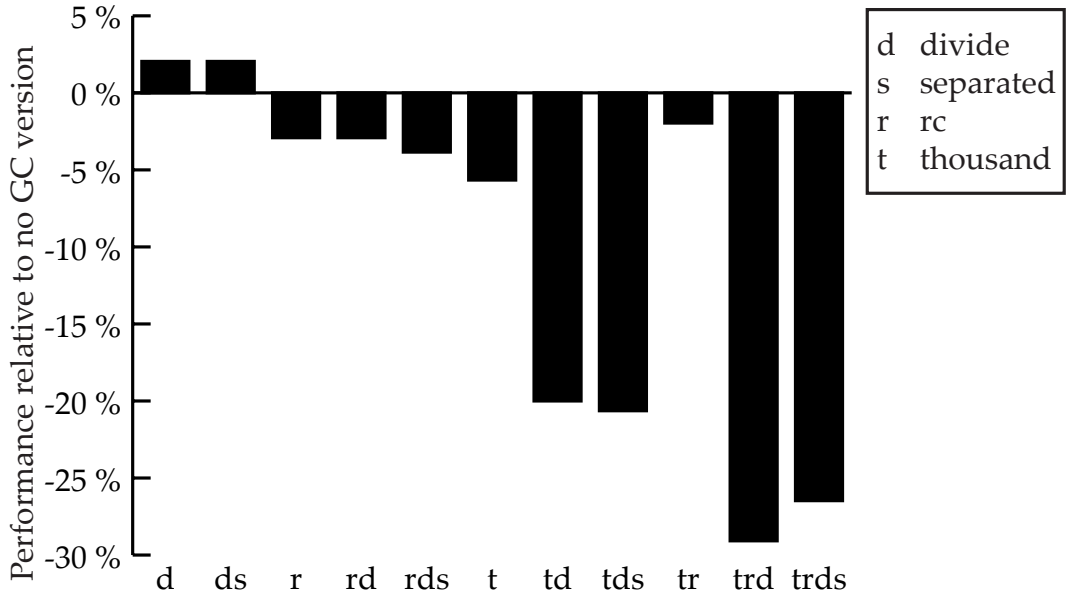


Figure 8.1. Control system benchmarks

8.2 Control System Application

To measure the impact of different memory usage profiles on a realistic hard real-time application, RTRC was tested on a control system simulation application. This test was written in C to get better control of the generated code. The control system was tested in 12 versions with and without: dividing objects in two blocks of 32 bytes each (d), separating the blocks of an object by half the heap size (ds), using reference counting (r), and running a thousand simultaneous simulations (t). The intent of running thousand simulations is to get a larger heap size and to get many cache misses. The result of this evaluation is presented in Figure 8.1 where the execution times of the different versions are compared to a system without garbage collection and without splitting the objects into blocks.

The conclusion from these tests is that dividing objects is expensive when cache misses are common. By comparing the reference counted variations to their non-reference counted counterparts, we get a maximum overhead of 13 %. The lowest overhead was achieved when comparing the test with thousand simulations without dividing objects to its reference counted counterpart. The reference counted version was 4 % faster. Over 24 million increment and decrement operations were performed in each run of the system (with reference counting enabled.)

8.3 Java Grande Benchmarks

The sequential Java Grande benchmark suite [BSW+00] was selected to measure the performance of JoC within Joses. Thus, efforts were focused on compiling these benchmarks, and still all benchmarks did not compile. Due to the instability of the compiler and the fact that the development has been discontinued, this benchmark suite is the only suite that has been used to benchmark the JoC RTRC implementation .

The benchmark applications were compiled in four versions: without GC, with GC and contiguous (i.e. not divided) objects, with GC and 32-byte blocks, and with GC and 64-byte blocks. The results are presented as performance improvements compared to the non-GC version.

8.3.1 Low Level Operations

The first section of the Java Grande sequential benchmarks tests low level operations such as: arithmetic, assignment, casts and object creation. Most of these applications should not be touched by the garbage collector at all. To further complicate the interpretation of the results, these applications can be optimized very aggressively using constant propagation and dead code elimination.

The measurements of the arithmetic applications are presented in Figure 8.2. Excluded from the diagram are the tests of addition of ints that gave a slowdown of 22 % and addition of longs that gave an improvement of 85 % – 178 %. The object creation tests failed to compile, and has thus also been excluded. The large variations in the integer addition applications are likely due to hard optimizations and differences in using the cache. The remaining test results show a variation of ± 5 %, which is normal due to variations in the data layout and cache usage. The integer addition and int and float multiplication benchmarks gave more than one operation per clock cycle which is a clear indication of heavy optimizations.

Next the assignment operations are measured. The results from these tests are presented in Figure 8.3, where the diagram excludes the results of the object instance assignments that gave a performance loss of about 99 %. The total loss of performance can only be explained by the fact that the optimizer could not eliminate the assignments. Thus, it is clear that this optimizer should be run before the reference counter, which would have improved the results drastically since many assignments in the benchmark are redundant. There are some remarkable results. First, assignment of local variables of primitive types are slowed down about 35 % in all reference counted version even though no GC-code is output in the timed code. Second, when testing assignment of class variables of primitive types, the contiguous version gave a slow down of 12 % – 17 % while the versions us-

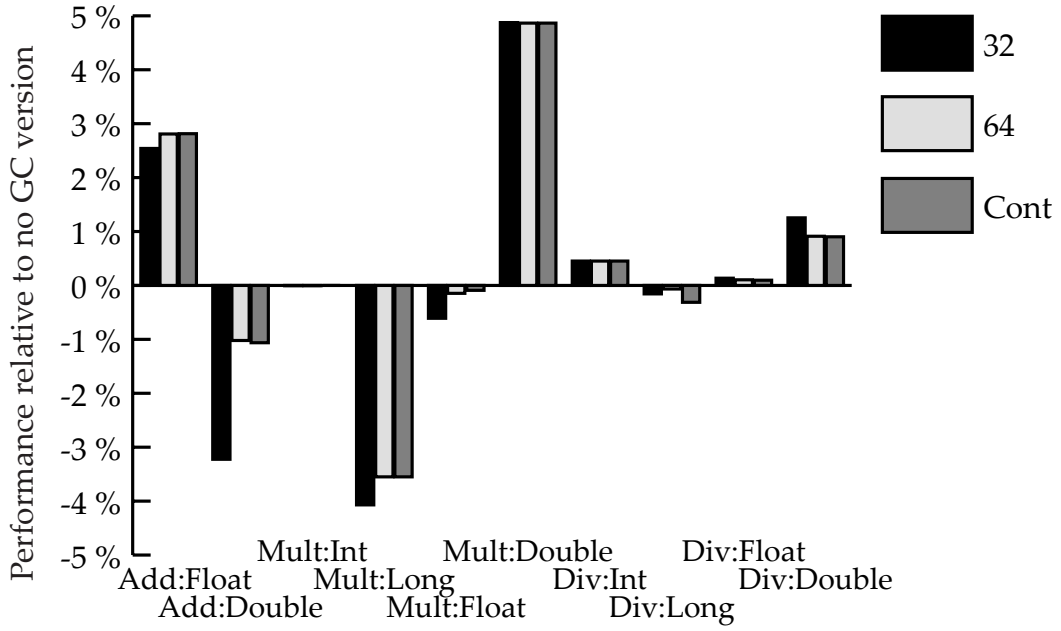


Figure 8.2. Java Grande — Section 1, Arithmetics

ing blocks gave a slowdown of 0 % – 4 %. Note that the class variables are allocated statically without using block structure. Third, assignment of array elements in class variables gave a slowdown of 12 % using blocks when the source element and target element were in difference classes, but when they were located in the same class there was no performance difference. Finally, assignment of elements in a local array showed a performance gain of 10 % when using RTRC. Again, the most likely explanation is differences in data layout, cache usage, and optimization. The remaining tests showed practically no difference in runtime performance.

Finally, cast measurements are presented in Figure 8.4. All measurements show a slowdown of 1 % – 9 % with an average of 6 %. As before, the measured code is not touched by the garbage collector.

This first section indicates the great difficulty in benchmarking a new technique. Even though the measured part of the application is not modified by the garbage collector, the results can show an improvement of performance with up to 178 %.

8.3.2 Kernels

Section 2 of the Java Grande sequential benchmarks contains various computationally heavy applications, such as IDEA-encryption, FFT and matrix multiplication. Most of these applications make little use of objects and therefore cause only a small amount of garbage collection work. The tests are executed in two versions with different sizes of the input. The results

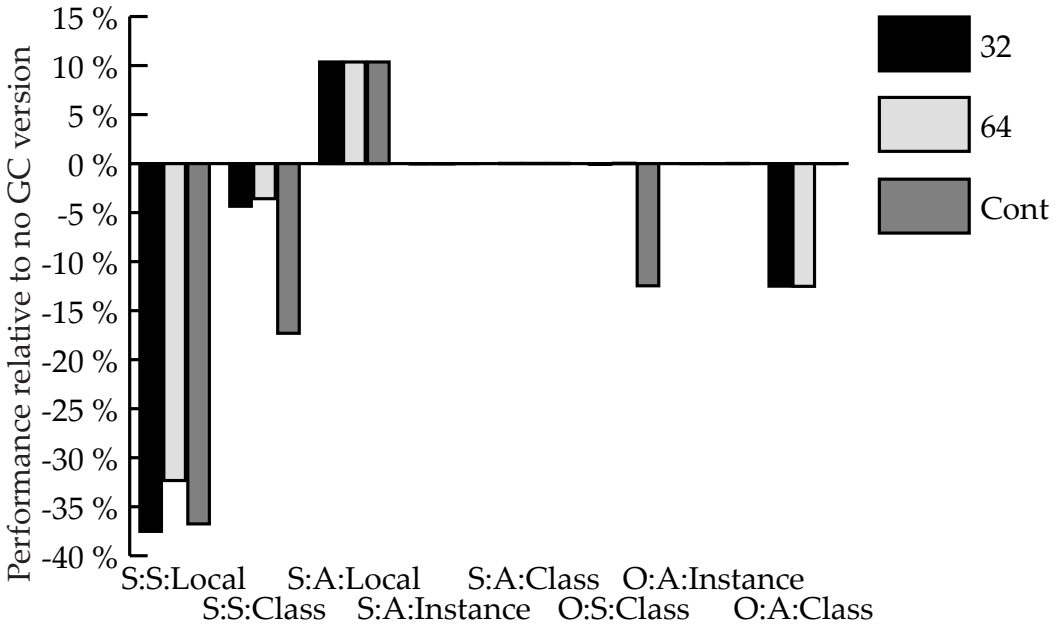


Figure 8.3. Java Grande — Section 1, Assignments

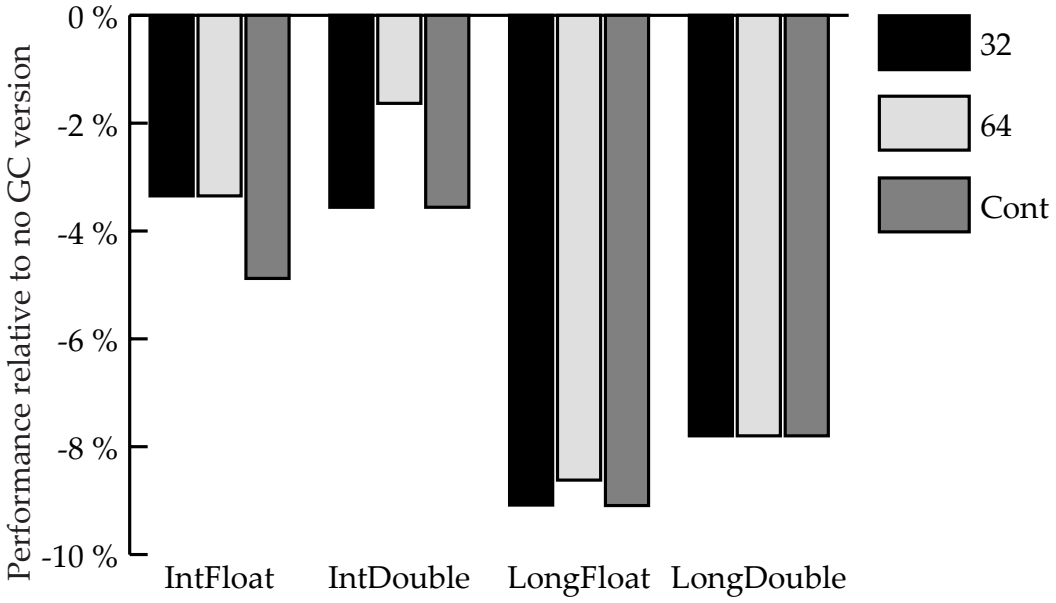


Figure 8.4. Java Grande — Section 1, Casts

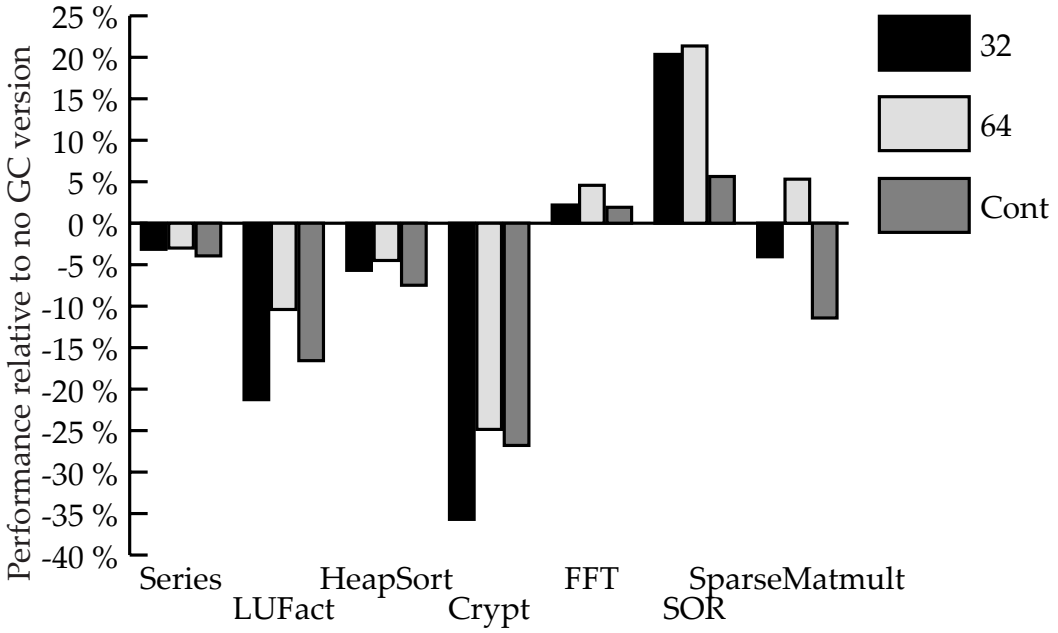


Figure 8.5. Java Grande — Section 2, Kernels, Size A

for the different inputs are similar. Three tests showed larger changes in performance: LUFact gave a slowdown of about 15 %, Crypt gave slowdown of about 30 %, and SOR gave a speedup of 20 %. The results are presented in Figure 8.5 and 8.6.

8.3.3 Large Scale Applications

The JOSES compiler JOC had some difficulties compiling the applications in section 3 of the Java Grande Benchmarks. The applications are object-oriented and perform computational heavy calculations such as alpha beta search and ray-tracing. Only the alpha beta search binary could be executed with a valid result, but MolDyn and RayTracer where executable so their benchmarks are also presented. As in section 2, two different inputs with different size were used, but only MolDyn could execute without a garbage collector with the larger input. The results are presented in Figure 8.7 and 8.8. The overhead in this section is significant, which can be explained by the amount of redundant reference count updates that is generated for all temporary reference variables that JOC generates. As noted in Section 8.3.1, this could be improved first by optimizing the code before the reference counter processed the code. The overhead can also be significantly reduced by static analysis.

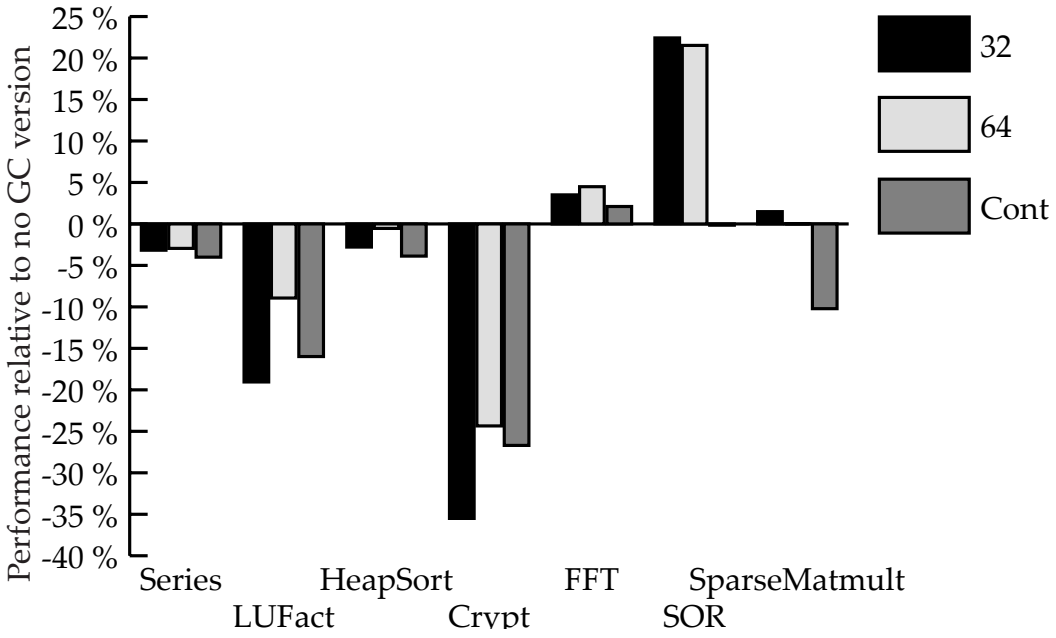


Figure 8.6. Java Grande — Section 2, Kernels, Size B

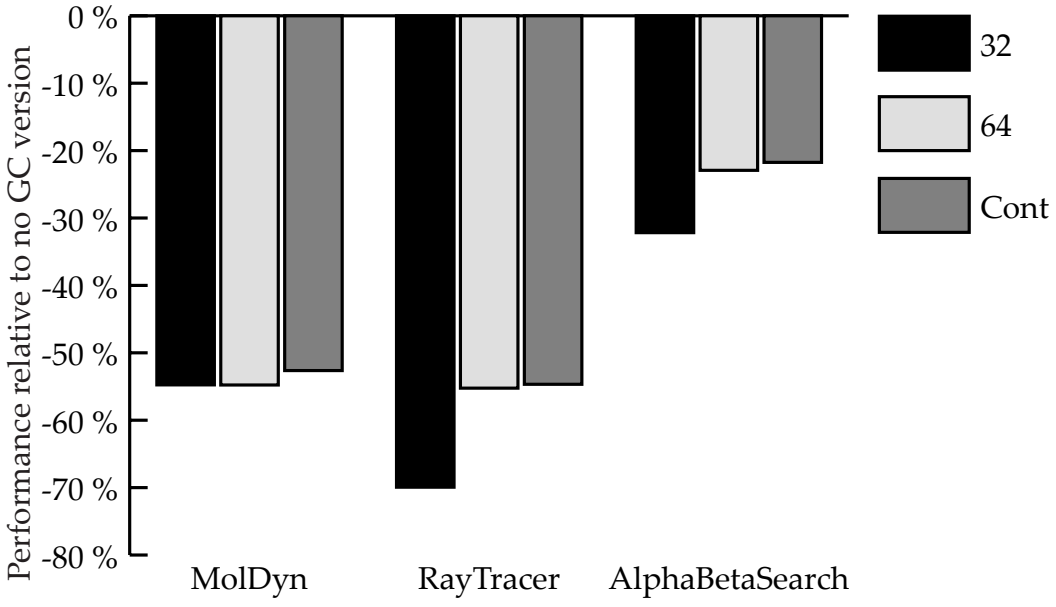


Figure 8.7. Java Grande — Section 3, Large Scale Applications, Size A

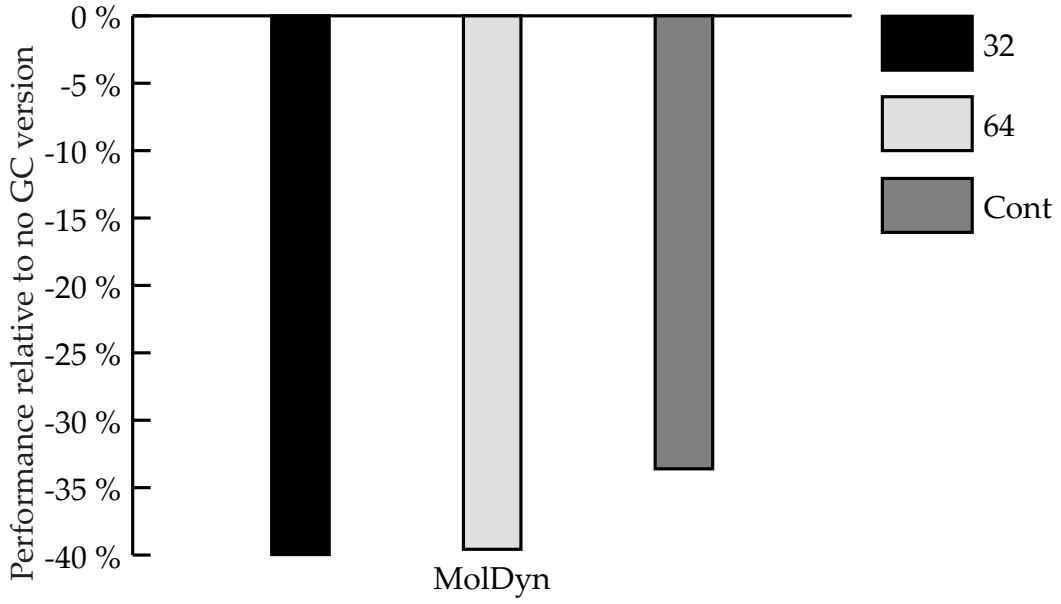


Figure 8.8. Java Grande — Section 3, Large Scale Applications, Size B

8.4 Summary

The control system application that was implement in C with full control of the generated GC code lost at most 29 % performance while making heavy use of reference counting and accessing fields in different blocks without using either static garbage collection or the object ownership optimization. The large scale applications in Java Grande showed a loss of 70 % performance in the worst case.

Much GC work could probably be eliminated by using optimizations that remove redundant assignments. Unfortunately, these optimizations could not process the code before the reference counter was invoked since the reference counter expected a higher level IR that the optimizers could not work on. It is likely that general applications loose 10 % – 25 % using optimized RTRC and control system applications that are written from scratch can be designed to loose less than 5 % using static garbage collection, object ownership, and static allocation of arrays. However, due to the effects of the cache, these results may vary.

— History will be kind to me
for I intend to write it.

Sir Winston Churchill

Chapter 9

Related Work

This chapter presents related work in the areas of real-time garbage collection, garbage collection optimizations, and static garbage collection.

9.1 Real-Time Garbage Collection

When this work started, there was no garbage collector available that was predictable in both memory usage and execution time and that targeted object-oriented languages. The one-pass mark-and-sweep collector, presented in Section 9.1.1, has a predictable execution time and is fairly easy to schedule to guarantee memory availability, but it only targets languages with no destructive operations, i.e. data can not be overwritten. Thus, it can not be used in the systems we target.

Still, several garbage collectors claimed to be real-time. Many incremental garbage collectors are presented as real-time collectors, e.g. Baker's copying collector that neither guarantees worst case interruption time nor memory availability. In some cases garbage collectors are even called real-time because the longest measured interrupt was shorter than a specified time, which is simply not acceptable in real-time systems.

During our work, three real-time garbage collectors have been presented: Henrikssons's copying garbage collector [Hen98], Siebert's mark-and-sweep collector [Sie02], and the mostly non-copying collector by Bacon et al. [BCR03]. These are presented below.

9.1.1 One-Pass Real-Time Mark-and-Sweep

Armstrong and Virding have proposed a one-pass mark-and-sweep garbage collector [AV95]. The collector is designed for languages that prohibit destructive operations (i.e. values can not be overwritten), and is used in

a Erlang [AVW93] implementation. Since destructive operations are not allowed, references can only refer to older objects. The collector marks the objects from the youngest to the oldest. If no other object has referred to an object that is being marked, it can be reclaimed immediately (since only younger objects can refer to it.) All objects are kept in a singly linked list to keep the objects in order. All GC operations have predictable runtime behavior. Thus, it can be used for real-time systems if it is scheduled to guarantee memory availability.

The collector can also work as a generational collector (see Section 3.6) by stopping the collector before it has collected the entire heap. It is even possible to run several collectors simultaneous on the same heap.

A compacting version [LF99] has been developed by Larose and Feeley. This collector compacts the objects instead of keeping the objects in a linked list. Unfortunately this modification causes the collector to loose its real-time properties.

However, the one-pass mark-and-sweep collector can not be used in systems where destructive operations are allowed.

9.1.2 Henriksson's Scheduling Strategy

Henriksson has presented a scheduling strategy for real-time garbage collectors [Hen98]. The tasks are divided into two categories, one with high priority and one with low priority. Both categories can be internally scheduled using any pre-emptive scheduling technique, so several priority levels are supported. The difference of the high and low priority tasks are that the high priority tasks only performs minimal garbage collection work. All tasks are predictable in both execution time and memory usage.

The main idea is to reserve and initialize the maximum amount of memory needed by an invocation of any high priority task, so high priority tasks are relieved of garbage collection work. However, some work is still required to maintain the garbage collection state when the object graph is modified. When leaving a high priority task to execute a low priority one, the work to reserve and initialize memory for the next invocation of a high priority task starts (see Figure 9.1.) This work can either be scheduled immediately when leaving the high priority task or it can be spread out during the execution of the low priority tasks. While low priority tasks are executing, the reserved memory is maintained by advancing the collector every time new memory is allocated. The amount of garbage collection work needed can be computed at compile-time. The required analysis is also provided by Henriksson.

Henrikson provides an implementation using CPP macros expressed in C. The implementation uses Brooke's version of Baker's copying technique (see Section 3.5.2) with modifications that include the new scheduling strategy. To relieve high priority tasks of garbage collection work

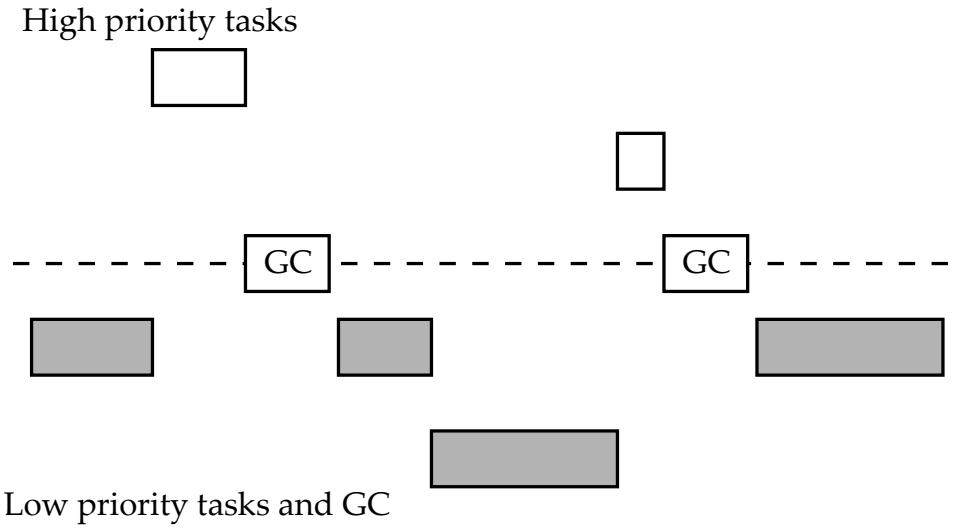


Figure 9.1. Using Henriksson's scheduling, garbage collection work is mostly done when leaving high priority tasks.

they do not copy any memory themselves. Instead memory regions that should be copied are queued and copied while low priority tasks execute. Since Brooke's algorithm uses two sub-heaps, more than three times the maximum amount of live memory is often required to keep the rate of copying down and the performance up. As a solution to this problem Henriksson proposes to use a mark-and-compact technique instead, but no such implementation has been presented. A mark-and-compact solution is further discussed in Section 10.1.3. However, since both copying and mark-and-compact solutions move objects, they suffer from long and hard to predict interrupts when large objects are moved.

9.1.3 Siebert's Real-Time Mark-and-Sweep

In parallel with our work, Siebert has presented a real-time mark-and-sweep collector [Sie02]. The heap is divided into equally sized blocks, and arrays are formed by connecting blocks into a tree structure. However, arrays can also be allocated contiguous if such memory is available. Other objects are formed using linked lists. The garbage collector does not collect objects. Instead individual blocks are reclaimed, which simplifies WCET analysis. The garbage collection work can be modeled as:

$$GC\ Work = Root\ Scanning + Marking + Sweeping$$

Scanning the roots is generally unpredictable, but Siebert solves it by copying all references on the stack to the heap and treats the blocks of refer-

ences as any block on the heap. The other roots are known at compile-time, therefore scanning the roots becomes predictable.

To improve efficiency, a threading model is developed that only allows thread switching at specified synchronization points. This model allows relaxations of the runtime behavior between the synchronization points, e.g. no locking is required during garbage collection operations, black objects (objects that have already been processed by the garbage collector) may refer to white ones (objects that has not yet been found by the garbage collector), and references on the stack may be ignored. However, at synchronization points all these requirements must still hold. For this to work, synchronization points must be spread out evenly in the code, so that the delay from trigger to invocation of a task can be predicted. Currently, this is not fully implemented, but synchronization points are emitted after a maximum amount of byte-codes, which is a solution that seems to work in practice. Note that this threading model only can be applied to single processor systems.

9.1.4 Mostly Non-copying GC

Bacon et al. have proposed a mostly non-copying technique that is designed to decrease the memory overhead without loosing execution time performance [BCR03]. The technique is based on an incremental mark-and-sweep collector that allocates equally sized objects in pages, which are not necessarily virtual memory pages. When a page becomes to fragmented, the objects on that page is copied to a mostly full page. To overcome the problem of locking the system while copying large objects, i.e. arrays, these are split into *arraylets*. Arraylets are fixed size arrays that are accessed via an indirection. Since they have a fixed size, the interruptions are bounded.

The presented benchmarks show a *minimum mutator utilization* (MMU) of 45 % while using 1.6 – 2.5 times the live memory without taking the object overhead into account. The MMU is defined as the minimum fraction of the processor devoted to mutator execution during a given time period. The object header contains type information and a forwarding pointer to find (as in Baker's copying algorithm), which occupies 8 bytes using a 32-bit architecture. Since objects may be moved, one word is needed to implement the `Object.hashCode()` method. When using non-moving collectors, this method normally returns the address of the object. The paper does not discuss the problem, but no better solution has been presented. For 22 byte objects (including the header), the overhead is $\frac{12}{12+22} \approx 35$ %. Thus, the memory usage of the system according to the model presented in Chapter 4 is about 26 % – 40 %. Note that these figures are measured and not worst case. The authors state that the worst case memory usage for most programs is smaller than those of a two sub-heap copying collector.

```
(defun five (x) (kill x) 5)
(defin square (x)
      (let* ((x x-prime (dup x)))
            (* x x-prime)))
```

Figure 9.2. Linear LISP examples

9.2 GC Optimization

9.2.1 Deferred and Anchored Pointers

The technique was presented by Baker [Bak94] for a LISP dialect called linear LISP. In Linear LISP each bound name is referenced exactly once, i.e. each actual argument or local variable must be used once. The `dup` function duplicates a value if it is to be used a second time (duplicating the reference does not count as a usage.) If a value is not used, it must be destroyed with the `kill` function. Two small examples are given in Figure 9.2.

A pointer can be marked as deferred, which means that it has a deferred reference count update, i.e. the reference count should have been incremented but the update has been deferred. If a deferred pointer is killed, the update is canceled against the decrement operation that would normally be executed. Several other rules must be applied for this technique to work correctly.

Deferred pointers can alone eliminate several reference count updates. However, a weakness is that only normal (non-deferred) pointers can be returned from functions. To get around this disadvantage, pointers can be anchored at a certain level in the stack. Below that level deferred pointer may be returned, but when returning to the level where it is anchored, the pointer must be converted to a normal pointer.

9.2.2 Reference Escape

An alternative approach to eliminate redundant reference count updates is presented by Park and Goldberg [PG95]. The idea of reference escape is the same as for object ownership: if an object is known to be kept alive by a reference, no temporary references to that object need to be counted. However, reference escape is designed for a functional language that has no side-effects, which is very different from object-oriented languages where side-effects are part of the foundation. The object ownership optimization can be viewed as an extension of reference escape that handles modern object-oriented languages.

9.3 Static Garbage Collection

Ruggieri first presented a partial life-time analysis for object-oriented languages [RM88]. The analysis associated each allocation with a function which can free the object when it returns. However, it is very restricted and does not support many of the features that are common in object-oriented programs, neither is any implementation mentioned. To decrease complexity, Park and Goldberg restricted the analysis to find data that can be allocated on the stack. This technique was called escape analysis.

9.3.1 Escape Analysis

Escape analysis [PG92] was first applied to functional programming. The technique has been incrementally improved and has also been applied to object-oriented languages. The research of escape analysis for object-oriented languages include [CGS+99, Bla99], which focus on eliminating synchronizations and allocating objects on the stack. Similar work is presented by Gay and Steensgaard [GS00], which also includes replacing objects by their fields stored as local variables. This work also mentions allocating objects in stack frames of the calling method. However, no presented work that we are aware of can associate allocations with functions that can free the allocated objects as we do. In this respect our static garbage collection is similar to the analysis presented by Ruggieri, but it can also be applied to modern object-oriented languages and an implementation is provided.

9.4 Region Interference

Similar to activation records, regions [TT93, Tof98] are stored on a stack, but regions allow objects to be added at any time in contrary to activation records. Thus, the same flexibility as dynamic allocation is obtained. However, individual objects can not be reclaimed from a region as they can be from the heap. The only way to reclaim an object is to reset the region that holds it. When a region is reset, all objects in the region are reclaimed.

Region interference analysis [TT93, Tof98] has been developed for functional languages. One implementation annotates Standard-ML programs with region information, so that all objects are allocated in regions. Thus, an annotated Standard-ML program needs no runtime garbage collector. Memory management is performed by allocating objects in regions, resetting regions, pushing regions on the stack, and popping regions from the stack.

The memory manager of our static garbage collector can be viewed as associating a region with every method invocation, and new objects are allocated into the region associated with the method that can reclaim it. However, the region interference analysis is designed for Standard ML like languages, and has currently not been applied to object-oriented languages.

— A thesis is abandoned, not completed.

Mikael Pettersson

Chapter 10

Future Work

10.1 Real-Time Runtime Garbage Collection

The static garbage collector presented in this thesis does its best to eliminate runtime garbage collection, but it can not cover all cases. The best runtime garbage collector is of course one that never has run, but we are far from that goal in the general case. There is still much work to be done within runtime collection for real-time systems.

10.1.1 Worst Case Memory Requirements Analysis

As stated before, by not knowing the memory usage of an application you face the risk of running out of memory. No runtime garbage collection will ever be able keep a system that uses too much memory up and running indefinitely. To overcome this problem, an analysis that calculates the worst case memory usage is required.

An interesting question is whether a runtime GC is needed when you have such an analysis. Is it possible to calculate how much memory a system uses without knowing where all allocated objects can be reclaimed? If it is impossible, the question is how to find out where objects can be reclaimed. Thus, this analysis is the same analysis needed by the static garbage collector. The reverse question is also interesting: Can you calculate the memory usage of a system if you know where objects can be reclaimed? If all objects are allocated statically or on the runtime stack, this is true for all systems where it is possible to calculate maximum recursion depth (and has a predictable object size.) However, what about systems that allocate from the heap?

10.1.2 Reclaiming Cycles

The reference counting technique can not reclaim cycles completely by itself. Section 4.5 presents several possible manual techniques that can be used, and in Section 4.6 an automatic technique is also presented. Manual techniques suffer from the human-factor problem and can cause memory leaks, and should be avoided if possible. The automatic alternative is to use a backup garbage collector. This has not yet been implemented in this work since it can be considered to be the last alternative, causing an overhead in both execution time and memory usage. Thus, both the manual and the automatic techniques have their disadvantages. Is there a better way of handling the dead cycles?

There are reference counting garbage collectors that are efficient in reclaiming cycles [BR01], but these are not designed for real-time. It would be very interesting to see if it is possible to extend such techniques without loosing too much performance.

10.1.3 Mark-and-Compact GC

The second best alternative to reference counting as the basis for our garbage collector is mark-and-compact. A mark-and-compact garbage collector marks the live objects as a mark-and-sweep collector, but instead of reclaiming the dead objects, the live ones are compacted. Again there is a problem when moving large objects, but with today's memory bandwidth a 1 MB object can be copied in about 2 ms. Therefore, such delays may not cause problems depending on the system that is designed. However, most embedded systems have significantly lower memory bandwidth (and usually smaller objects.) The benefit of using mark-and-compact is that you get fast access to large objects, with much improved memory usage compared to two sub-heap copying (about 45 % using the same model as in Section 4.10.2.) The drawback is that you do not know exactly when the garbage collector interrupts the system, since objects may be moved at every barrier (e.g. at every reference assignment.) Henriksson's scheduling analysis [Hen98] can be used to guarantee that the interrupts do not occur while a high priority task executes, but the start of a high priority task may still be delayed by the interrupt.

10.1.4 Optimizations

Further optimizations are always welcome, thus a hot topic for the future. It would be interesting to know how often object ownership can be used in real applications. The analysis required is similar to the static GC technique presented in Chapter 6. An integration of these techniques may be fruitful. It would also be interesting to see how much object ownership can increase the performance of RTRC and other incremental garbage collectors.

10.2 Static Garbage Collection

Most important to the static garbage collector is the ability to handle full systems, and to handle objects that are stored in fields of other objects. The design for these additions is presented in Section 6.6. The performance of systems using the static garbage collector can be further improved by extending the analysis. Some ideas are presented in the following sections.

10.2.1 Inter-Procedural Def-Use Analysis

Most garbage collectors keep objects alive for as long as they are reachable. However, many objects are reachable long after their last use. If an inter-procedural def-use analysis can be developed, it would be possible to reclaim these objects after their last use. The result could be used by the static garbage collector to explicitly reclaim the object or by a runtime garbage collector by explicitly marking the object as unreachable after its last use.

10.2.2 Reusing Objects in Loops

Objects that are allocated inside loops can reuse the same memory area if the objects die before the next iteration of the loop. The analysis is similar to the presented static GC technique, but control flow has to be taken into account as well. What is actually needed is to find out if the object can be used after the loop, so the inter-procedural def-use analysis described above may be used to determine whether the objects can share memory or not.

10.2.3 Object Inlining

Another application of the escape analysis is the possibility of inlining objects within other objects, i.e. instead of allocating separate objects; their fields can be merged into their parent object. This is possible when the lifetime of an object is confined within the life time of another object. This typically occurs when an object is accessed from exactly one other object. Another application of object inlining is to replace a stack allocated object by its fields as local variables.

10.2.4 Supporting Separate Compilation

Even without converting the graph into a tree, large system will take time to analyze. It would be preferable to support separate compilation to improve analysis time. Separate compilation is possible if the sub-graphs of methods are stored with the compiled code. The sub-graph should

preferably be condensed using the shortcuts described in Section 6.6.2. The graphs can be merged when analyzing the final system.

However, since a library is not a complete system, it is possible to find calls to unknown methods, such as methods in other libraries or callbacks. Calls to other libraries can be handled by including the other library in the analysis (possible using shortcuts in a separate analysis), but callbacks can not be handled like this.

10.3 Dynamically Updated Systems

Most inter-procedural analyses such as the ones used by the static garbage collector and the object ownership optimization assume that the full system is available at compile-time. However, modern languages have the possibility of loading updates and additions while the system is up and running [ACR98, AR00]. Is it possible to extend these analyses so that such hot-swaps can be allowed? One work around is to update the complete system and do the analysis on the complete system, and then update all affected code, but what if we do not know exactly what code is running on the system? That may be the case if users of different systems can upload new features. This problem is not isolated to garbage collection. The same problem appears in WCET analysis and many other global analyses.

Modern languages also support introspection (or reflection). Introspection allows a system to access type information, create objects, examine and change data, and invoke methods, using strings to specify classes, methods and attributes. Thus, it is easy to write an application that is virtually impossible analyze. Unfortunately, these features may be interesting to use even in real-time systems. The same problem may occur when using dynamically linked libraries. Is it possible to do any useful analysis of such systems?

The extreme of a dynamic system are systems that generate executable code, e.g. to handle regular expressions efficiently as implemented in C#. It is virtually impossible to analyze the generated source at compile time (of the generator), but it may be possible to do something before the code is executed. A possible approach is to integrate the analysis in the runtime system.

— Sometimes even a blind hen finds a grain of corn.

Unknown

Chapter 11

Conclusion

As the development of hardware progresses, computers are expected to solve increasingly complex problems. Solving more complex problems requires more complex software. To be able to develop these software systems, new programming languages with higher levels of abstraction are introduced. If we generalize the development of programming languages, they have evolved from an algorithmic description close to the machine representation towards a problem description closer to human thinking. This simplifies development of software, but also makes it harder to maintain control of the executed code, and thus over the runtime behavior of the software. Despite the fact that the hard real-time domain requires full control of the running software these modern languages become increasingly popular in real-time applications, since the development time is decreased and the software becomes less error prone.

Since the 80's most of the new popular languages have been object-oriented. However, object-oriented languages have a larger need for garbage collection than their imperative counterparts. The increased need for garbage collection stems from the object-oriented way of programming where it is common to dynamically allocate more objects (or records) compared to programming using the imperative paradigm. Since more objects are dynamically allocated, it becomes harder to handle memory management manually. To eliminate several memory management related programming errors, most modern languages lack the possibility to reclaim objects manually, and require garbage collection instead.

When this work was started, no presented garbage collection technique could guarantee predictability in both execution time and memory usage for the languages we target. A one pass mark-and-sweep GC [AV95] for functional languages that do not allow destructive operations had been published. That garbage collector, as well as other ones, is possible to schedule to guarantee memory availability, but the necessary analysis was

not presented. Thus, many techniques had predictable interrupt times, but they did not guarantee that the garbage collection work progressed fast enough for the system to never run out of memory.

The first approach that could guarantee memory availability was a two sub-heap copying algorithm [Hen98]. The scheduling technique is not specific for the copying technique, and can be used by other techniques as well. Systems that could not afford the overhead in execution time and memory usage had to be handled manually using static allocation and possibly stack allocation.

A second approach [Sie02] that guarantees both predictability in execution time and memory usage has been presented. This approach is based on mark-and-sweep garbage collection, and solves the external fragmentation problem by dividing the heap into equally sized blocks. The advantages of this approach are slightly higher memory utilization and that objects do not move around on the heap. However, dividing the heap into blocks has a cost when accessing data in objects that contain more than one block.

A third approach which is a hybrid solution of mark-and-sweep and copying garbage collection [BCR03] has been presented. The technique is designed to have low memory requirements, while keeping a consistent CPU utilization. However, the worst case memory usage is worse than the two sub-heap copying GC in some cases. A further discussion of these techniques and the cost of large objects is presented in Section 11.2.

The three previously presented techniques all suffer from low memory utilization. The memory utilization of the copying garbage collector is approximately 25 %, the mark-and-sweep collector can use approximately 30 % of the heap to store actual data, and the hybrid approach has similar memory requirements.

Another disadvantage of the techniques is that it is very difficult to estimate the memory requirements of applications that use them. The reason for the popularity of garbage collection is that the developer is not required to know when objects turn into garbage and can be reused, since manual memory administration is very hard and error prone. However, if it is not known when memory can be reused, it is also unknown how much memory can be in use at any given point of time. Thus, the memory requirements of applications are usually unknown. *It is, of course, possible to estimate the memory requirement of an application, but it is at least as error prone as manual memory management.* The problem is actually harder, since references can be forgotten and keep objects alive longer than expected. A common solution is to run the application and hopefully find the worst case memory usage this way. This can be compared to running the system to find its WCET. Sometimes this is acceptable, but for security critical systems it is not.

11.1 Garbage Collection in Real-Time

Due to the problems of predicting the run-time behavior of garbage collected systems, one should consider if garbage collection can at all be a solution for hard real-time systems. Some would say that even using dynamic memory is to complex. On the other hand some would not use while loops or recursion either [BW89]. Somewhere a line must be drawn.

Most real-time developers would use while loops and recursion if they can be proved to do what is expected. Dynamic memory management is more difficult, since this is much harder to prove correct. The problem is twofold: not only is it hard to reclaim memory correctly, external fragmentation (a problem that occurs when free memory is split into several small regions) may also cause the system to run out of memory even though memory is available.

We can overcome these problems using a garbage collector that does not suffer from external fragmentation. However, new problems are introduced by the garbage collector. The garbage collector must be predictable in both execution time and memory usage, i.e. it is not enough to prove that the garbage collector only interrupts the systems for a certain period of time, it must also be guaranteed that enough memory is reclaimed so that the system does not run out of memory.

There are garbage collectors (including real-time reference counting as presented in this thesis) that guarantee predictability in both execution time and memory availability. The problem is that the memory requirement of the system must be known to be able to guarantee its memory usage. No garbage collection technique can keep a system that uses too much memory from running out of memory. Thus, the problem is how to predict the memory requirement of a garbage collected system. This is a hard problem since there is no easy way of telling when memory can be reclaimed. If there were, no garbage collector would be necessary.

Whether it is safe to use a garbage collector in a hard real-time system or not depends on the complete system. If you can predict the worst case memory requirement, it is perfectly safe to use a correctly configured garbage collector that can guarantee memory availability. Otherwise, static garbage collection (e.g. as presented in this thesis) can be used to simplify the work of calculating the worst case memory requirements by inserting explicit free instructions and allocating objects statically and on the heap. If the memory requirements are still hard to predict, it might be good to rethink the design of the system. It will probably be even worse to predict the runtime behavior of such system if manual memory management is used.

11.2 Selecting a Base Algorithm

Since no garbage collector had predictable runtime behavior when this work started, we developed real-time reference counting (RTRC). The reason for basing the garbage collector on reference counting is that we found it most suitable after comparing the existing techniques.

Two sub-heap copying garbage collectors have the advantages of compacting the memory and that its execution time is only dependent on the amount of live memory. However, moving objects around in memory is time consuming and can cause long interruptions in the system. It also makes it necessary to access the objects via handles to find the latest versions. The solution of updating all references to point to the new object gives better access times, but updating all references can be very time consuming even if their locations are known.

To cut down on the interruption time, you may allow the copying operation to be interrupted, but then you have to restart the copying of that object from the beginning to guarantee that the latest version of the object is used. This causes problems in guaranteeing progress, and thus memory availability. Another problem is that only half the heap can be used to store live objects, which causes large memory requirements.

More sub-heaps can be used to increase the average performance of a copying garbage collector. Such techniques are usually referred to as generational garbage collectors. More sub-heaps may also increase memory utilization, since only one sub-heap needs to be empty using some techniques (while other techniques split each generation into sub-heaps.) Unfortunately the worst case execution time of generational garbage collectors are often worse than the corresponding operations using two sub-heap copying techniques. Even worse is the fact that cyclic dependencies can keep dead objects alive for long times. This makes predicting memory requirements extremely hard.

Mark-and-sweep garbage collection can be used to avoid copying objects. However, since objects are not compacted external fragmentation has to be considered. The memory requirements of mark-and-sweep garbage collectors are also substantial. However, a mark-and-sweep garbage collector is a good candidate as a backup garbage collector to RTRC.

A combination of copying and mark-and-sweep garbage collectors is called mark-and-compact garbage collectors. These use less memory than two sub-heap copying collectors and compacts the memory. Unfortunately, they share the problem of long interrupts and guaranteeing progress. Still, mark-and-compact is a good alternative and could probably give good results in both memory usage and execution time.

All presented techniques have problems with large objects. For techniques that copy objects, large objects cause long interrupts or problems in guaranteeing progress. Current non-copying techniques solve the prob-

lem of external fragmentation using equally sized blocks. To access data in large objects, several dereferences are required which consumes additional time. Large objects, such as bitmaps and large arrays, are best allocated statically irrespective of what technique is used.

11.3 Contributions

The contributions of this work can be divided into three parts. First, the work on a real-time garbage collector that was presented in our previous work [Rit99, Rit01, RF02] has continued. The resulting technique is called *Real-Time Reference Counting* or RTRC for short. Since reference counting is slow, a new optimization technique called *Object Ownership* was developed that eliminates many counting operations (and thus memory accesses) for object references with limited life-span. Finally an *improved static garbage collector* was developed to eliminate runtime garbage collection when possible and to simplify the work of estimating the memory requirements of applications.

11.3.1 Real-Time Reference Counting

The major advantage of RTRC over other real-time garbage collection techniques is that its runtime behavior is independent of the amount of live memory in the system (as long as the memory is not exhausted.) Thus, the memory utilization increases by about 50 % compared to the most memory efficient approach (mark-and-sweep). The memory utilization of RTRC is compared to the memory usage of the other presented real-time garbage collectors in Figure 11.1.

Another advantage is that locking is kept to only a few assembler instructions. Thus, the interruptions caused by the garbage collector can be ignored if the real-time responsiveness of the system is not in the magnitude of nanoseconds. Furthermore, the extended escape analysis can be used to eliminate locking for objects that are only used by a single thread. Finally, to improve performance of high priority processes, the scheduling analysis technique by Henriksson [Hen98] can be used. The main impact of using the scheduling technique is that high priority tasks perform less garbage collection work, which improves the performance of allocation.

11.3.2 Object Ownership

Since RTRC is based on reference counting, which is a relatively slow garbage collection technique, we developed an optimization, called Object Ownership, which increase the execution time performance of the write-barrier in RTRC with up to 75 %, or eliminates the write-barrier completely,

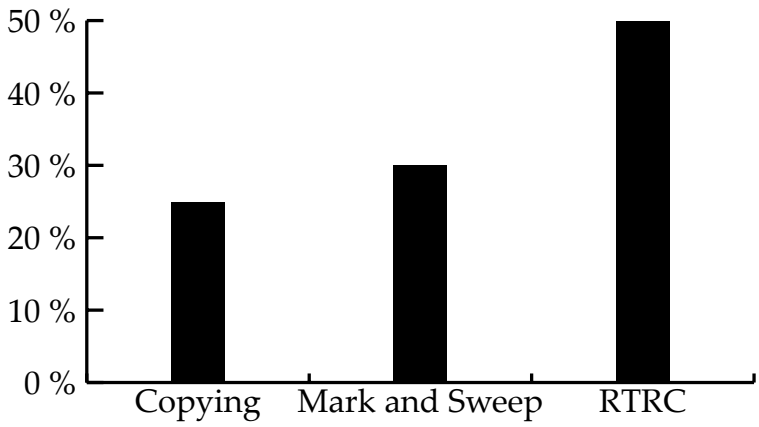


Figure 11.1. Available memory for memory usage in % of total used memory when using RTRC compared to using other real-time garbage collectors

wherever the technique can be applied. It optimizes the garbage collector by finding objects that are known to be kept alive by a reference. This reference is said to own the object. Other references that refer to the object while it is owned by another reference can be ignored by the garbage collector, and the overhead caused by counting these references can be eliminated.

Object ownership was designed for reference counting, but can also be adapted to other incremental garbage collection techniques to decrease the overhead of barriers (i.e. code that is executed when objects are accessed to update the garbage collector state.)

11.3.3 Static Garbage Collection

To eliminate runtime GC and to simplify estimation of memory requirements, a new static garbage collection technique was developed. The current state of the art of static garbage collection methods with reasonable analysis time is limited to stack allocation, whereas the analysis presented here also can handle objects that require heap allocation. The only objects that can not be handled by this technique are the objects that correspond to the global state of the application, i.e. the objects that may stay alive during all of the execution of the application, and objects that are used by multiple threads. If these objects are allocated statically, there is no need to have a runtime garbage collector in the system. No static garbage collector previously presented in the literature can do this with a reasonable analysis time.

11.3.4 Usefulness of Contribution

RTRC does not completely replace the competitive techniques, since RTRC has some problems when it comes to cyclic data structures. Even though both manual and automatic approaches to handle cyclic data structures are presented in Chapter 4, some systems may contain too much cyclic data structures and be too complex to handle manually to benefit from RTRC. Since the performance gain of RTRC is in the allocator, applications that rarely allocate objects will gain less than applications that allocate objects more frequently. The main use of RTRC is in real-time systems with scarce memory resources, typically embedded systems.

The object ownership technique can be useful to all incremental garbage collectors by eliminating redundant barriers. The technique can be used in a conservative mode where fewer barriers are removed, or in a more aggressive mode that requires extra data in the object header.

Finally the static garbage collector can be used to boost performance and to simplify the work of predicting runtime behavior. The latter is only useful when memory requirements are important. However, boosting the performance by decreasing the overhead of runtime garbage collection or eliminating it completely, is beneficial in all systems using garbage collection.

Appendix A

Source Code of the Reference Counter (C version)

A.1 rc.h

```
#ifndef RC_H

#ifndef NUM_BLOCKS
#  define NUM_BLOCKS 10000
#endif

#ifndef BLOCK_SIZE
#  define BLOCK_SIZE 32
#endif

#define TOBLOCK(N) (((N) + (BLOCK_SIZE-1))/BLOCK_SIZE)

#define RCHEAD_NEXT(OBJ)   (((objhead_t *) (OBJ))->next)
#define RCHEAD_FLNEXT(OBJ) (((objhead_t *) (OBJ))->nr.next)
#define RCHEAD_RC(OBJ)     (((objhead_t *) (OBJ))->nr.rc)
#define RCHEAD_TYPE(OBJ)   (((objhead_t *) (OBJ))->type)

typedef void object_t;

struct objhead_t;

typedef struct type_t {
  char *name;
  size_t size;
```

```
  void (*decchildren)(struct type_t *t, struct objhead_t *b, int n);
} type_t;

typedef struct objhead_t {
  struct objhead_t *next;
  union { struct objhead_t *next; unsigned int rc; } nr;
  type_t *type;
} objhead_t;

extern void      rc_init();
extern object_t *rc_alloc(type_t *t);
extern object_t *rc_hpalloc(type_t *t);
extern void      rc_prealloc(int n);
extern void      rc_release(objhead_t *obj);
extern long      rc_stat_inc, rc_stat_dec, rc_stat_incc, rc_stat_decc;

#ifdef STAT
#  define RC_STAT(X) (X)++
#else
#  define RC_STAT(X)
#endif

#ifndef RC

#define RC_INCR(OBJ)
#define RC_DECR(OBJ)

#else

#define RC_INCR(OBJ)                              \
  do {                                            \
    if (OBJ) {                                    \
      RCHEAD_RC(OBJ)++;                           \
      RC_STAT(rc_stat_incc);                      \
    }                                             \
    RC_STAT(rc_stat_inc);                         \
  } while(0)

#define RC_DECR(OBJ)                              \
  do {                                            \
    if (OBJ) {                                    \
      RCHEAD_RC(OBJ)--;                           \
      RC_STAT(rc_stat_decc);                      \
      if (RCHEAD_RC(OBJ) == 0) {                  \
        rc_release((objhead_t *)OBJ);             \
      }                                           \
    }                                             \
    RC_STAT(rc_stat_dec);                         \
```

```
  } while(0)

#endif

#define RC_ASSIGN(LHS, RHS)                          \
  do {                                               \
    register void *_rtrc_tmp = (RHS);                \
    RC_INCR(_rtrc_tmp);                              \
    RC_DECR(LHS);                                    \
    (LHS) = (_rtrc_tmp);                             \
  } while (0)

#define RC_FUNCALL(RES,FUNC)                         \
    do {                                             \
        void *_rtrc_tmp = (rcobject_t *)RES;         \
        (RES) = (FUNC);                              \
        DECRC(_rtrc_tmp);                            \
    } while(0)

#define RC_RETURN(RES)                               \
    do {                                             \
      ASSIGN(result,(RES));                          \
      goto _cleanup;                                 \
    } while (0)

#endif
```

A.2 rc.c

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

#include "rc.h"
#include "debug.h"

#define CPS 100

#define MIN(A,B) (((A)<(B))?(A):(B))

long rc_stat_inc, rc_stat_dec, rc_stat_incc, rc_stat_decc;

static objhead_t *freelist  = NULL;
static size_t     available = 0;
static objhead_t *tbflist   = NULL;
static char       heap[NUM_BLOCKS][BLOCK_SIZE];
static objhead_t *head      = NULL;
```

```
static type_t    *type      = NULL;
static int        blockseq  = 0;

void rc_init() {
  int i;

  DEBUG_MSG1(("rc_init\n"));

#ifdef SPREAD
  for (i = 0; i < NUM_BLOCKS/2; i++) {
    int upper = NUM_BLOCKS/2 + i;
    ((objhead_t *) heap[i])->next = (objhead_t *) &heap[upper];
    ((objhead_t *) heap[upper])->next = (objhead_t *) &heap[i+1];
  }
#else
  for (i = 1; i < NUM_BLOCKS; i++) {
    ((objhead_t *) heap[i-1])->next = (objhead_t *) &heap[i];
  }
#endif
  ((objhead_t *) heap[NUM_BLOCKS-1])->next = NULL;
  freelist = (objhead_t *) heap;
  available = NUM_BLOCKS;
}

static objhead_t *
alloc_from_freelist(int nb, objhead_t **plast) {
  objhead_t *bs = freelist;
  int n;

  assert(nb > 0);
  assert(available >= nb);

  available -= nb;

  for (n = 1; n < nb; n++) {
    freelist = freelist->next;
  }

  {
    objhead_t *last = freelist;
    freelist = freelist->next;
    last->next = NULL;

    *plast = last;
  }

  return bs;
}
```

```
static objhead_t *alloc_from_TBF(int nb, objhead_t **plast) {
  objhead_t *bs = head, *last = NULL;
  int n;

  assert(nb > 0);

  DEBUG_MSG1(("alloc from tbf %d\n", nb));
  for (n = 0; n < nb; n++, blockseq++) {
    if (head == NULL) {
      DEBUG_MSG1(("New object from tbf\n"));
      head = tbflist;
      if (head == NULL) {
        fprintf(stderr, "Out of memory %d:%d:%d\n",
                n, nb, blockseq);
        exit(5);
      }

      tbflist = tbflist->nr.next;
      type = head->type;
      blockseq = 0;

      if (last == NULL) {
        bs = head;
      } else {
        last->next = head;
      }
    }

    type->decchildren(type, head, blockseq);
    last = head;
    head = head->next;
  }
  last->next = NULL;
  *plast = last;

  return bs;
}

void rc_prealloc(int nb) {
  objhead_t *bs, *last;

  assert(nb > 0);

  DEBUG_MSG1(("rc_prealloc %d\n", nb));

  if (available >= nb) return;
```

```
  bs = alloc_from_TBF(nb - available, &last);

  available = nb;
  last->next = freelist;
  freelist = bs;
}

object_t *rc_alloc(type_t *t) {
  int nb;
  int fromfree;
  objhead_t *blocks, *tbfblocks = NULL;
  objhead_t *last, *dummy;

  assert(t != NULL);

  nb = t->size;
  fromfree = MIN(nb, available);

  DEBUG_MSG1(("rc_alloc %s (%d/%d)\n",
              t->name, fromfree, nb - fromfree));

  if (nb - fromfree > 0) {
    tbfblocks = alloc_from_TBF(nb - fromfree, &dummy);
  }

  if (fromfree > 0) {
    blocks = alloc_from_freelist(fromfree, &last);
    last->next = tbfblocks;
  } else {
    blocks = tbfblocks;
  }

  blocks->type = t;

  return blocks;
}

object_t *rc_hpalloc(type_t *t) {
  int nb;
  objhead_t *blocks;
  objhead_t *dummy;

  assert(t != NULL);

  nb = t->size;

  DEBUG_MSG1(("rc_hpalloc %s (%d)\n", t->name, nb));
```

```
  blocks = alloc_from_freelist(nb, &dummy);
  blocks->type = t;

  return blocks;
}

void rc_release(objhead_t *b) {
  DEBUG_MSG1(("releasing: %p (%s)\n",
              b, RCHEAD_TYPE(b)->name));
  b->nr.next = tbflist;
  tbflist = b;
}
```

Appendix B

Source Code of the Object Ownership Test

B.1 ootest.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define OWN
#define OWNINI NULL
//#define FORCEFLD

#ifdef OWN
#define FORCEFLD
#endif

#define N 1000000
#define N2 5

#ifdef OWN
#define GC(P) ((P) != NULL && (P)->owner == NULL)
#else
#define GC(P) ((P) != NULL)
#endif

#define RELEASE(P) do { (P)->next = fl; fl = (P); } while (0)

#define INC(P) do { if(GC(P)) (P)->rc++; } while (0)

#define DEC(P)                                          \
  do {                                                  \
```

```
    if(GC(P) && --(P)->rc == 0) RELEASE(P);        \
  } while (0)

typedef struct foo {
  struct foo *next;
  int rc;
#ifdef FORCEFLD
  void *owner;
#endif
  int data;
} foo;

foo *fl = NULL;

int main() {
  int i, j, s = 0;
  foo *fs = malloc(N*sizeof(foo));

  clock_t start, stop;

  for (i = 0; i < N; i++) {
    fs[i].rc = 10;
#ifdef OWN
    fs[i].owner = (void *) OWNINI;
#endif
  }

  start = clock();
  for (j = 0; j < N2; j++) {
    for (i = 0; i < N; i++) {
      int r1 = (int) ((double) rand()*(N-1)/RAND_MAX);
      int r2 = (int) ((double) rand()*(N-1)/RAND_MAX);
      foo *f1 = &fs[r1];
      foo *f2 = &fs[r2];

      INC(f1);
      DEC(f2);

      s += r1 + r2;
    }
  }
  stop = clock();

  printf("%.2f (%d)\n", (double) (stop-start)/CLOCKS_PER_SEC, s);

  return 0;
}
```

Bibliography

[ACE] ACE Associated Compiler Experts bv, Amsterdam, The Netherlands. *The CoSy Compilation System*. see `http://www.ace.nl/products/cosy.htm`.

[ACR98] Jesper Andersson, Marcus Comstedt, and Tobias Ritzau. Runtime support for dynamic Java architectures. In *Proceedings of the Workshop on Object-Oriented Software Architectures*. The ECOOP'98 Workshop on Object-Oriented Software Architectures. Brussels, July 1998.

[AKPY98] Alain Azagury, Elliot K. Kolodner, Erez Petrank, and Zvi Yehudai. Combining card marking with remembered sets: How to save scanning time. In Jones [Jon98], pages 10–19.

[Alm97] Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 32–59. Springer-Verlag, New York, NY, 1997.

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[AR00] Jesper Andersson and Tobias Ritzau. Dynamic code update in JDrums. In *Procedings of the First workshop on Software Engineering for Wearable and Pervasive Computing (SEWPC) in Conjunction with ICSE'2000*, Limerick, June 2000.

[AV95] Joe Armstrong and Robert Virding. One-pass real-time generational mark-sweep garbage collection. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Computer Science Laboratory, Ellemtel Telecommunications Systems Labo ratories, Alvsjo, Sweden, September 1995. Springer-Verlag.

[AvSM93] U. Aßmann, H. van Someren, and Alt M. Compilers for Parallel architectures – The Compare project. In Sips H.J., editor, *Fourth International Workshop on Compilers for Parallel Computers*, volume 786, pages 451–454. Delft University of Technology, Faculty of Applied Science, Advanced School for Computing and Imaging, December 1993.

[AVW93] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

[Bac97] D. F. Bacon. *Fast and Effective Optimzation of Statically Typed Object–Oriented Languages*. PhD thesis, University of California, 1997.

[Bak78a] Henry G. Baker. Actor systems for real-time computation. Technical Report MIT Rep. TR–197, Laboratory for Computer Science, March 1978.

[Bak78b] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[Bak92] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[Bak94] Henry G. Baker. Minimising reference count updating with deferred and anchored pointers for functional data structures. *ACM SIGPLAN Notices*, 29(9), September 1994.

[Bar77] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[BC92] Yves Bekkers and Jacques Cohen, editors. *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 September 1992. Springer-Verlag.

[BCR03] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[Ben90] Mats Bengtsson. *Real-time Compacting Garbage Collection Algorithms*. Licentiate thesis, Department of Computer Science, Lund Un iversity, 1990.

[Ben98] Joakim Bengtsson. Compiling Java bytecode. Master's thesis, Linköping University, 1998.

[BJMM02] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 153–164, Berlin, June 2002. ACM Press.

[Bla99] Bruno Blanchet. Escape analysis for object oriented languages. application to Java. In OOPSLA [OOP99], pages 20–34.

[BR01] David F. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In Jørgen Lindskov Knudsen, editor, *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001*, volume 2072 of *Springer-Verlag*, Budapest, June 2001. Springer-Verlag.

[Bre89] R. P. Brent. Efficient implementation of the first-fit strategy for dynamic storage allocation. *ACM Transactions on Programming Languages and Systems*, 11(3):388–403, July 1989.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, August 1984. ACM Press.

[BSW+00] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, (12):375–388, 2000.

[BW72] F. L. Bauer and H. Wössner. The Plankalkül of Konrad Zuse: a forerunner of today's programming languages. *Communications of the ACM*, 15(7):678–685, 1972.

[BW89] Alan Burns and Andy Wellings. *Real-Time Systems and their Programming Languages*. International Computer Science Series. Addison-Wesley, 1989.

[CB01] Perry Cheng and Guy Belloch. A parallel, real-time garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 125–136, Snowbird, Utah, June 2001. ACM Press.

[CGS+99] Jong-Deok Choi, M. Gupta, Maurice Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In OOPSLA [OOP99], pages 1–19.

[Che70] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[Chr84] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.

[CJ73] Liu C.L and Layland J.W. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[Com97] Marcus Comstedt. Natural sematics specification and compiler generation for Java. Master's thesis, Linköping University, 1997.

[Con00] J Consortium. Real-time core extensions for the Java platform, 2000.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[DH01] Sylvia Diekmann and Urs Hölzle. The allocation behavior of the specjvm98 Java benchmarks (extended version). In Rudi Eigenman, editor, *Performance Evaluation and Benchmarking with Realistic Applications*. MIT Press, 2001.

[DLM+76] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[DLM+78] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[ESL89] H. Emmelmann, F.W. Schröer, and R. Landwehr. Beg — a generator for efficient back ends. *ACM Sigplan Notices*, 24(7):227–237, 1989.

[Fis74] David A. Fisher. Bounded workspace garbage collection in an address order preserving list processing environment. *Information Processing Letters*, 3(1):25–32, July 1974.

[fJEG00] The Real-Time for Java Expert Group. *The Real-Time Specification for Java*. Addison Wesley Professional, 2000.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction (CC'2000)*, volume 1781 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[HD90] Richard L. Hudson and Amer Diwan. Adaptive garbage collection for Modula-3 and Smalltalk. In Eric Jul and Niels-Christian Juul, editors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[Hen98] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Bekkers and Cohen [BC92].

[Hol00] Mikael Holmén. Natural semantics specification and frontend generation for Java 1.2. Master's thesis, Linköping University, 2000.

[HW67] B. K. Haddon and W. M. Waite. A compaction procedure for variable length storage elements. *Computer Journal*, 10:162–165, August 1967.

[Jon98] Richard Jones, editor. *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, October 1998. ACM Press.

[Kar00] Henrik Karlsson. Exception handling in RT-Java. Master's thesis, Linköping University, 2000.

[KS77] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120–131. IEEE Press, 1977.

[LF99] Martin Larose and Marc Feeley. A compacting incremental collector and its performance in a production quality compiler. *ACM SIGPLAN Notices*, 34(3):1–9, 1999.

[LG85] I. Lee and V. Gehlot. Language constructs for distributed real-time systems. In *Proceedings of the Real-time Symposium*. IEEE Computer Society Press, 1985.

[LH83] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[Lia99] Sheng Liang. *Java Native Interface*. Addison-Wesley, 1999.

[MR94] James S. Miller and Guillermo J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report AIM-1462, MIT AI Laboratory, March 1994.

[NO93] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993. ACM Press.

[NOPH92] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Bekkers and Cohen [BC92].

[NS93] Kelvin D. Nilsen and William J. Schmidt. Cost-effective object-space management for hardware-assisted real-time garbage collection. *Letters on Programming Languages and Systems*, 1(4):338–354, December 1993.

[OOP99] *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, Denver, CO, October 1999. ACM Press.

[Pet95] Mikael Pettersson. *Compiling Natural Semantics*. PhD thesis, Linköping University, 1995.

[PG92] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the 5th ACM SIGPLAN conference on Programming language design and implementation*, pages 116–127. ACM Press, 1992.

[PG95] Young G. Park and Benjamin Goldberg. Static analysis for optimising reference counting. *Information Processing Letters*, 55(4):229–234, August 1995.

[RBLP00] Tobias Ritzau, Marcel Beemster, Florian Liekweg, and Christian Probst. JoC — the JOSES compiler. Presented at the Java for Embedded Systems Workshop, London, May 2000.

[RF02] Tobias Ritzau and Peter Fritzson. Decreasing memory overhead in hard real-time garbage collection. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, volume 2491 of *Lecture Notes in Computer Science*, pages 213–226. Springer, 2002.

[Rit99] Tobias Ritzau. *Real Time Reference Counting in RT-Java*. Licentiate thesis, Linköping University, March 1999.

[Rit01] Tobias Ritzau. Hard real-time reference counting without external fragmentation. In Dr. Uwe Assmann, editor, *Java Optimization Strategies for Embedded Systems Workshop at ETAPS 2001*, Genova, Italy, April 2001.

[RM88] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–293. ACM Press, 1988.

[Roj98] R. Rojas. How to make Zuse's Z3 a universal computer. *IEEE Annals of the History of Computing*, 20(3):51–54, 1998.

[Roo00] Anders Roos. Class and object representation in the RT-Java compiler. Master's thesis, Linköping University, 2000.

[RSLR95] Ragunathan Rajukumar, Lui Sha, John P. Lehoczky, and Krithi Ramamritham. An optimal priority inheritance policy for synchronization in real-time systems. In Sang H. Son, editor, *Advances in Real-Time Systems*, chapter 11, pages 249–272. Prentice Hall, 1995.

[Sau74] Robert A. Saunders. The LISP system for the Q–32 computer. In E. C. Berkeley and Daniel G. Bobrow, editors, *The Programming Language LISP: Its Operation and Applications*, pages 220–231, Cambridge, MA, 1974. Information International, Inc.

[Sch88] William F. Schmitt. The UNIVAC SHORT CODE. *Annals of the History of Computing*, 10(1):7–18, January/March 1988.

[SG95] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In O. Nierstras, editor, *Proceedings of 1995 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 235–252, University of Aarhus, August 1995. Springer-Verlag.

[Sha88] Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Stanford University, 1988. Technical Report CSL–TR–88–351.

[Sie02] Fridtjof Siebert. *Hard Realtime Garbage Collection*. PhD thesis, Universität Karlsruhe, 2002.

[spe98] Spec jvm98 benchmarks. Available at `http://www.specbench.org/osg/jvm98/`, 1998.

[SRL88] Liu Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical report, Department of Computer Science, Carnegie-Mellon University Pittsburgh PA, 1988.

[Ste75] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[Ste76] Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley Professional, 1994.

[TCD86] Alan Turing, B.E. Carpenter, and R.W. Doran. *A. M. Turing's ACE Report of 1946*. MIT Press, 1986.

[TGJS96] Java Team, James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Tof98] Mads Tofte. A brief introduction to Regions. In Jones [Jon98], pages 186–195.

[TT93] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report Computer Science 93/15, University of Copenhagen, July 1993.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[Vee01] Arthur Veen. The JOSES project - compiling Java for embedded systems. In Dr. Uwe Assmann, editor, *Java Optimization Strategies for Embedded Systems Workshop at ETAPS 2001*, Genova, Italy, April 2001.

[vN45] John von Neumann. First draft of a report on the EDVAC. Technical report, Moore School of Electrical Engineering of the University of Pennsylvania, 1945.

[Wei63] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.

[WJ93] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.

Index

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity

of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN

91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-suppor-ted Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationsystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering - att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.