Marta Stojanovic

AUTOMATIC MEMORY MANAGEMENT IN JAVA

Mémoire
présenté
à la Faculté des études supérieures
de l'Université Laval
pour l'obtention
du grade de maître ès sciences (M.Sc.)

Département d'informatique
FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL

JUILLET 2001

0-612-65392-7

À ma famille

Remerciements

Mes études à l'Université Laval n'auraient pas été les mêmes sans mes nombreux amis. Même s'il y en a qui n'étaient pas directement liés à ma maîtrise, leur présence m'a beaucoup aidé et facilité ma vie d'étudiante. Et comme cette partie de ma vie finit maintenant, je trouve que ce chapitre est presque la seule opportunité de les remercier.

Je tiens premièrement à remercier mon directeur de recherche Mourad Debbabi, qui savait me pousser au bon moment et sans qui je n'aurais sûrement pas terminé ma maîtrise. Merci pour votre support, pour nous avoir fait chanter en revenant de Montréal, pour m'avoir appris comment écrire des messages électroniques, pour écrire des lettres de référence si poétiques et pour me laisser faire des blagues avec vous.

Je dois aussi remercier Nadia Tawbi, Jules Desharnais et Jean Bergeron, que j'appelle "les trois mousquetaires" : ce qui a eu une note humoristique, mais aussi parce que le travail qu'ils faisaient était très noble. Ils prenaient soin de nous, étudiants de M. Debbabi, lors de son absence, et ils faisaient cela avec beaucoup de plaisir et volonté. Je vous remercie infiniment. En particulier, je dois remercier Madame Nadia Tawbi, pour avoir accepté d'être mon évaluatrice, et pour partager avec moi ses pensées et son bonheur de nouvelle mère.

Je remercie Mohammed Mejri, à l'époque mon collègue, avec qui j'avais du plaisir à travailler sur le projet DYMNA, pour avoir accepté d'être mon évaluateur et pour être un très bon ami.

Merci à Darko Stefanovic du Département d'informatique de l'Université du Nouveau Mexique et Ole Agesen de VMWare Inc., que je ne connais pas personellement, mais qui m'ont beaucoup aidé, par la correspondance électronique, à comprendre les problèmes liés aux les systèmes de ramasse-miettes.

Pendant mes études j'ai rencontré beaucoup d'étudiants, et beaucoup d'entre eux ont eu un bon impact sur moi. Je vous remercie tous infiniment :

Myriam Fourati, une fille exceptionnelle, pour être une si bonne amie, pour avoir des bons conseils sur n'importe quel sujet de la vie, pour m'avoir toujours supporté et pour m'avoir enduré dans le laboratoire à côté d'elle pour tout ce temps.

Mouetsie Molière, pour être toujours là pour moi, pour être prête à danser et sortir avec moi en tout temps, pour me laisser des messages drôles, pour m'apprendre les danses latinos, et pour m'avoir intéressée à la musique chorale.

Lamia Ktari, pour être toujours souriante et prête à faire des blagues, pour être l'organisatrice invisible de toutes les fêtes surprises du labo, pour avoir toujours le temps de m'écouter et pour m'aider à chaque fois que je le lui ai demandé (et même sans demander).

Therrezinha Fernandez, pour son sens d'humour et pour supporter mes bêtises pendant un gros travail que nous avons fait ensemble.

Yolaine Légaré et Nadine Côté, avec qui j'ai travaillé sur le projet DYMNA, pour m'avoir supporté pendant les jours et les nuits de travail sous une pression énorme. Un grand merci à Nadine, qui était ma première amie durant mes études à la maîtrise.

Béchir Ktari, qui est un logo du groupe LSFM et qui était un support pour tous les étudiants du groupe, pour savoir répondre à mes nombreuses questions techniques, pour des longues conversations au sujet de la situation politique dans le monde, pour avoir joué au tennis avec moi, et pour avoir été courageux d'essayer de danser le swing avec moi.

Bogdan Chalupa, avec qui je faisais une excellente équipe dans mes cours probatoires du baccalauréat, pour tout ce qu'il m'a appris, et pour être devenu un ami pour la vie.

Mark Girard, "le macho man", pour ses blagues quotidiennes qui me manquent énormément, pour sa gentillesse et pour nous avoir montré comment être un parent modèle.

Stéphane Doyon, un vrai génie, pour me rapprocher de Linux, pour son aide et sa gentillesse, et juste pour sa présence et son énergie positive.

Kamel Adi, pour m'avoir supporté constamment et pour ses précieux conseils.

Mourad Erhioui pour être toujours prêt à aider et pour me rapprocher des méthodes formelles.

Frédéric Massicote pour m'avoir toujours supporté pendant notre projet DYMNA et pour m'avoir transmis une petite partie de sa passion pour les sports.

François Daigneault et Stéphane Larose pour avoir rendu mes pauses dîner beaucoup plus intéressantes, pour m'avoir laissé entrer dans leur vie, et pour me rajeunir un peu.

Emmanuel Giasson pour tester ma compréhension du français québécois et pour sa grande gentillesse.

Et mes autres amis, pour m'avoir aidé et partagé leurs pensées avec moi ou juste pour être là : Zaineb Ben Fredj, Yvan Lavoie, Hamdi Yahyaoui, Hakim Chambaz, Denis Davesne, Frédéric Painchaud, Frédéric Michaud, Daniel Huot, Luc Poulin, Vincent Labbe, Emna Menif, Elhachemi Alikacem ... Il en reste sûrement beaucoup que j'ai oubliés. Je suis désolée si c'est le cas, c'est juste une question d'âge.

J'ai eu la chance d'avoir des très bons enseignants au Département de l'informatique de l'Université Laval (et, malheureusement, je n'ai pas eu la chance de suivre tous les cours qui me semblaient intéressants). J'ai déjà mentionné quelques enseignants, mais il y en a d'autres. Je pense que je n'oublierai jamais les excellents cours de Mourad Debbabi et Denis Laurendeau, c'est vraiment un grand plaisir d'assister aux cours offerts par les gens qui ont un talent pour l'enseignement comme ces professeurs. On oublie souvent les chargés de cours, mais plusieurs fois il arrive qu'ils donnent de meilleurs cours que certains professeurs. Je tiens à remercier en particulier Abderrahim Alikacem, Jacques Despatis et Louis Bastarache, qui sont vraiment de très bons enseignants et en même temps très disponibles et très proches des étudiants. En plus, M. Bastarache et son cours m'ont ouvert la porte vers le travail que je fais maintenant, et je le remercie en particulier pour cela.

Le Département d'informatique de l'Université Laval a un excellent support technique. Durant ma maîtrise, j'ai travaillé surtout sur Unix et Linux, et j'aimerais remercier Denis Poirier et Louis Demers pour m'avoir installé des milliers de logiciels, pour répondre avec plaisir à un grand nombre de mes questions et pour être si disponibles.

Je ne dois pas oublier les dames très gentilles du secrétariat, qui étaient toujours souriantes et prêtes à aider : Lynda Goulet, Doris Laneuville-Ouellet, Rachel Légaré-Lapierre et Anne Gosselin. Je vous remercie pour tout.

Peut-être il n'est pas encore temps de remercier les gens de CNRC, où je travaille présentement, mais il y en a parmi eux sans qui cette maîtrise ne serait pas la même. Je tiens à remercier premièrement mon superviseur Khaled El-Emam, pour m'avoir poussé à finir et pour me donner tout ce temps et support pour le faire. J'aimerais remercier Roger Champagne, travailleur invité au CNRC et un très bon ami, pour avoir corrigé mes textes en français (celui-ci inclus) et pour être toujours prêt à m'aider. Je n'oublierai jamais les conseils et le support que j'ai reçus de Margaret Dalziel, je la remercie énormément. Shadia Elgazzar et Moshe Krieger m'ont toujours supporté pendant la rédaction de mon mémoire et la préparation de ma présentation, et je les remercie également. Merci à Charles-Antoine Gauthier pour avoir miraculeusement sauvé mes données à plusieures reprises. Et finalement, je dois remercier mon chef d'équipe Anatol Kark, pour avoir été patient avec moi et m'avoir accordé du temps pour travailler sur la maîtrise.

Enfin, je pense que les gens qui m'ont aidé et qui ont participé le plus à ma maîtrise sont les gens dans l'ombre : ma famille. Mon mari Milos, qui m'a toujours supporté dans mon cheminement et qui a enduré les jours et les nuits que j'ai passés à étudier à l'université ou à la maison, le stress qui allait avec chaque examen ou travail pratique et les hauts et les bas de ma recherche ; mon fils Andrej, qui manquait maman souvent durant les fins de semaine et les soirées ; ma mère, ma belle-mère, mon père et ma sœur qui sont venus de la Yougoslavie pour m'aider en gardant Andrej, en nettoyant, en cuisinant et en me supportant, de l'aide que je n'oublierai jamais. Je vous remercie de tout mon cœur, sans vous je ne serais jamais arrivée au bout.

Résumé

Dans ce travail, nous essayons d'analyser l'allocation automatique de mémoire en Java. Nous présentons d'abord une étude sur des algorithmes de base d'allocation de mémoire et de ramasse-miettes. Puis, nous présentons des résultats de tests de six systèmes de ramasse-miettes implantés dans différentes machines virtuelles de Java. Nous avons constaté qu'un simple système de ramasse-miettes marque-balaye ("mark-sweep") dans LaTTe JVM donne des résultats égaux ou meilleurs que des systèmes très sophistiqués qui implantent un ensemble d'algorithmes de ramasse-miettes dans les machine virtuelles récentes de Sun. Afin d'expliquer cette excellente performance, nous avons analysé le code du système d'allocation de mémoire de LaTTe. Ceci nous a permis de mieux le comprendre et d'identifier les dispositifs qui augmentent son efficacité. Nous suggérons des améliorations possibles à l'algorithme ainsi que des algorithmes alternatifs qui semblent intéressants à implanter et analyser. À l'avenir, nous voudrions implanter les améliorations et algorithmes proposés pour la gestion de mémoire dans LaTTe et comparer leur efficacité à l'algorithme actuel.

Abstract

In this work, we try to analyze the automatic memory management in Java. We first present a survey on basic allocation and garbage collection algorithms. Then, we present benchmark results on six garbage collectors implemented in different Java virtual machines. We unexpectedly found that a rather simple mark-and-sweep garbage collector in LaTTe JVM performs equally well or better than highly sophisticated collectors that implement a set of garbage collection algorithms in recent Sun's Java implementations. In order to explain this excellent performance, we reverse engineered the code of LaTTe's memory management system, which enabled us to better understand it, to find the features that add to its efficiency, to suggest possible improvements on the present algorithm and other garbage collection algorithms interesting for implementation and testing. In the future, we would like to be able to implement suggested improvements and algorithms for memory management in LaTTe and to compare their efficiency to the present one.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Automatic Memory Management

Dynamic memory management [1] is an integral characteristic of every modern programming language. There are two types of dynamic memory management: manual and automatic. Manual memory management requires the programmer to explicitly return memory to the language when it is no longer needed. Automatic memory management (or *garbage collection*) frees the programmer from this burden; memory is automatically reclaimed when the run-time system can determine that it can no longer be referenced. Manual memory management is used in C, C++, Pascal, Ada, and Modula II. Automatic memory management is used in LISP, Scheme, Eiffel, Modula III, and Java. Several reasons lead to the integration of the automatic memory management into the programming languages. We will mention some of them: complexity and size of applications, size of language data structures, object-oriented programming.

Complexity and size of applications As a limited resource, memory has always been a great concern to programmers. At the beginning of computer science, when the memory was very expensive, it was necessary to carefully allocate and deallocate the memory. Programs were simple and predictable and the memory was explicitly allocated by programmer or compiler. As the price of the memory went down, the applications became larger and more memory demanding. More complex programs lead to more complex problems of deallocation of memory, and the need for an automatic memory allocation arose.

Size of language data structures There are languages that use rather simple data structures, and manual memory management is sufficient to handle them. Other languages typically manipulate large data structures with complex inter-dependencies.

[1] The memory management of an application is said to be dynamic if its storage needs vary during the execution.

Functional and logical languages have complex patterns of execution. Automatic memory management seems necessary in those languages.

Object-oriented programming Object-oriented analysis and programming is a rapidly developing area in computer science. The keyword of object-oriented approach is encapsulation of objects that communicate through clearly defined interfaces. Manual memory management contradicts this modularity by dispersing the code for memory deallocation throughout the application code. For this reason most modern object oriented languages support garbage collection.

1.1.2 Java

Among the languages that use automatic memory management, Java (developed by *Sun Microsystems, Inc.*) is the most popular today. It is an object-oriented language, but also an imperative, parallel and distributed language. Java gained popularity thanks to its mobile code: it permits the mobility of code through the Internet. Java code is compiled to a binary code, called *bytecode*, which is recognized by a Java virtual machine (JVM). A JVM acts as an interface between the compiled Java binary code and the microprocessor (or "hardware platform") that actually executes the program's instructions. Once a Java virtual machine has been provided for a platform, any Java program (bytecode) can run on that platform. Java was designed to allow application programs to run on any platform without having to be rewritten or recompiled by the programmer for each separate platform. Thus, Java virtual machines make Java portability possible. Furthermore, The Java Application Programming Interface (API) is rich in classes and easy to use. Java has also a reputation of being a safe language, with its exception handling, security policies and garbage collection. All these characteristics make Java very popular today.

1.1.3 Garbage Collection and Java

Why did Java implement a garbage collector? There are several reasons and we have already mentioned some of them. One of the security aspects of a programming language is good memory management. If it is left to the programmer, there is a high chance to create either a dangling reference or a garbage (see details in Section 2.1.2), which can lead to either premature lack of memory or an incorrect reference. Garbage collection takes care of all these problems. Furthermore, there is a belief that an object-oriented language should have garbage collection implemented. Indeed, Meyer places automatic memory management in third place in his list of "seven steps to object-based happiness" [1]. Sun's Java illustrates the need for garbage collection in object-oriented languages. This project originally chose C++ as the implementation language, but the difficulties encountered with C++ grew over time to a point where the engineers felt that problems could be overcome only by designing a new language, Java. One important feature lacked by C++ but included in Java is a garbage collector. Java has brought garbage collection to the mainstream, being the first truly popular language in the C/C++ tradition that requires garbage collection.

Garbage collection can be implemented in several ways, and the reason why the research in this field is still active is that there is no perfect algorithm: one algorithm can be excellent for one kind of application and bad for another, depending on the allocation behavior of an application. Sun, for instance, changed several times the garbage collector algorithm in its JVMs in order to have a collector that is more efficient, having a satisfactory performance for more application types. First versions of Java Development Kit had a partly conservative compacting mark-sweep collector. The newest Sun's JVM (Hotspot), has an accurate generational garbage collector, which is copying in the first generation and either marc-compacting or incremental in the second generation (see Section 3.2.1 for detailed description). Sun claims that, between these two collectors, the latter is much more efficient. The literature on garbage collection supports this claim, but there are some programmers who found that for some applications the older algorithm works better. It seems that a run-time decision on garbage collection algorithm is necessary to satisfy all possible applications.

1.2 Objectives

The objectives of this work are to:

- understand the principles, the advantages and the disadvantages of basic memory management algorithms;
- test the existing JVMs with several carefully chosen applications in order to make the assumptions on the efficiency of the implemented garbage collection algorithms:
- fully understand the garbage collector's interface in a Java virtual machine of our choice, by a thorough analysis of its code (we chose LaTTe JVM ([2]) based on its performance and availability);
- discuss possible improvements of existing LaTTe's memory management system, as well as a suitable garbage collection algorithm that could replace the existing one, without having to change the structure of the virtual machine.

1.3 Organization

This document is organized as follows : Chapter 2 gives a review of the existing literature on memory management algorithms (with an accent on garbage collection). Chapter 3 presents the evaluation of the efficiency of garbage collector systems in various JVMs. The results of the reverse engineering of LaTTe's memory manager code are presented in Chapter 4. Chapter 5 presents a discussion on LaTTe's memory management system, and its possible improvements. Finally, Chapter 6 concludes.

Chapter 2

Memory Management Algorithms

Our main objective was to study the most commonly used garbage collection algorithms. But memory management consists of both allocation and deallocation, and therefore we will present both allocation and deallocation techniques, with the accent on garbage collection. Much of this chapter is based on the following excellent reviews: Wilson's review on memory allocation [3], Jones and Lins' book on garbage collection [4] from which the actual algorithms are taken and Wilson's uniprocessor garbage collection review [5].

2.1 Introduction

In this section we present a short history of storage allocation, the reason for the development of automatic memory management (garbage collection) algorithms, its disadvantages, as well as the issues that have to be considered when comparing the garbage collection algorithms.

2.1.1 History of Storage Allocation

With the apparition of high-level programming languages, a compiler needed to allocate resources of the target machine to represent the data objects manipulated by the user's program. There are three ways in which storage can be allocated: static, stack and heap allocation.

In the case of *static allocation* all the names in the program are bound to storage locations at compile-time and these bindings do not change at run-time. Main disadvantages of this type of allocation are that the size of each data structure must be known at compile-time, that no procedure can be recursive and that data structures cannot be created dynamically.

In *stack allocation* an activation record or frame is pushed onto the system stack as each procedure is called, and popped when it returns. The characteristics of this allocation are: recursive calls are possible, the size of local data structures such as arrays may depend on a parameter passed to the procedure, a called activation record cannot outlive its caller and only an object whose size is known at compile-time can be returned as the result of a procedure.

Heap allocation allows data structures in a heap to be allocated in any order (not necessarily last in - first out). With this type of allocation, allocation records and dynamic data structures may outlive the procedure that created them, the size of data structures can vary dynamically, dynamically-sized objects may be returned as the result of a procedure and result of a function can be a data structure which outlives the activation of the function that created it.

Today most high-level programming languages are able to allocate storage on both the stack and the heap. The allocation is explicit (the programmer allocates the memory needed), but memory reclamation can be both explicit (the programmer is responsible for freeing unused objects) or implicit, or automatic (garbage collector searches and frees objects that are no longer used). Many languages (Pascal, C, C++) manage all data on heap explicitly. Functional, logic and most object-oriented languages use garbage collection to manage the heap automatically (Scheme, Dylan, ML, Haskell, Miranda, Prolog, Smalltalk, Eifell, Java, Oberon). Modula-3 offers both explicitly and automatically managed heaps.

2.1.2 Reasons for Garbage Collection

The need for garbage collection arose from the problems with explicit deallocation, but also from the language and problem requirements, as well as some software engineering issues. Unfortunately, garbage collection is not a universal solution.

Problems with explicit deallocation

Manual memory management can cause the apparition of garbage and dangling references which can lead to the obstruction of the running program :

Garbage Dynamically allocated storage may become unreachable. For example, the head of a linked list can be made to point to nil. In that case, all the other members of the list become unreachable, and cannot be deallocated. Such objects, that are not alive, but are not free either, are called garbage. With explicit deallocation garbage cannot be reused; the corresponding space has *leaked* away.

Dangling references If the second member of a linked list is deallocated, the third member (together with the rest of the list) will become garbage like in the case above. However, at the same time, the pointer from the first member of the list to the second (deallocated) member will still refer to memory that has been deallocated and can possibly be reallocated: a *dangling reference* has been created.

Garbage collection avoids both of these problems.

Other reasons for garbage collection

Garbage collection may be a *language requirement.* If data structures may survive the procedure that created them, then it may be impossible for the programmer or compiler to determine the point at which it is safe to deallocate them.

Garbage collection may be a *problem requirement.* If a stack is implemented as a linked list and the pop operation deallocates the first element (the top) of the stack, should it deallocate the data referenced from this element ? The answer really depends on whether the data is statically allocated or not, whether it is the last pointer to the data etc.

Garbage collection may be a *software engineering issue.* Abstraction and modularity are among the key principles of the software engineering. Explicit memory management is not compliant with these principles. Automatic memory management gives increased abstraction to the programmer, the model of memory allocation is less low-level, so that programmers are relieved of the burden of book-keeping detail. As for modularity, it should not be necessary to understand an entire program before being able to develop a single module. And as liveness is a global property, changes to book-keeping code might have an influence beyond the module being developed.

Not a universal solution

The garbage collection is not a perfect solution for the memory management. Programs with straightforward dynamic memory requirements may be supported at *lower run-time cost* by explicit deallocation (although the short-term gain may have a longer-term cost). Memory demands of *hard real-time systems* cannot yet be solved without a hardware support. Garbage collection has *its own costs*, in terms of both time and space. Although garbage collection removes dangling pointers and space leaks, it is vulnerable to other errors. Garbage collection has no solution for the problem of *data structures that grow without bound* (such as caching the intermediate results to avoid recomputation). Tracing garbage collectors identify live data by following pointers from the roots of the computation including the program stack. But the stack can become polluted by obsolete pointers and if these pointers are traced the *space leaks* might occur.

2.1.3 Comparing Garbage Collection Algorithms

Many garbage collection algorithms have been proposed and it is rather difficult to compare them. The reason is that the main principles that should be taken into account when comparing the algorithms (cost of reclaiming cells, allocation cost. overhead on user program operations, delays caused by garbage collector, the amount of memory required for the collector etc.) depend on different parameters. These algorithms are usually tested on different machines, with different processor and on different operating systems, the implementation of one algorithm could influence its performance, the topology and volume of live data on the heap can influence the execution time, the order in which a graph is traversed or copied may effect the virtual memory behavior.

2.2 Basic Allocation Mechanisms

Although our work is concerned with garbage collection algorithms, in order to understand the memory management as a whole (which will be especially needed when analyzing the source code of a chosen memory management system, see Chapter 4), we will briefly mention the mostly used allocation mechanisms. This section is based on a review on dynamic storage allocation by Wilson et al. [3].

The main job of an allocator is to keep track of which parts of memory are in use, and which are free. An ideal allocator would spend negligible time managing memory and waste negligible space. The main problem is an application program, which can free objects in any order, creating holes in memory. If these holes are numerous and small, than they cannot be used for larger blocks. This problem is known as *fragmentation*.

Dynamic allocation has been studied for several decades, but it is still rather unknown field. In fact, it has been proven that any allocation algorithm in some cases can perform very badly causing a severe fragmentation, but some allocation algorithms have been shown in practice to work well with real programs and are widely accepted.

We will briefly present the following basic allocator mechanisms :

- *sequential fits*, including first fit, next fit, best fit, and worst fit,
- *segregated free lists*, including simple segregated storage and segregated fits,
- *buddy systems*, including conventional binary, weighted, and Fibonacci buddies, and double buddies, and
- *bitmapped fits*.

2.2.1 Sequential Fits

In this kind of allocator, all free blocks are linked in one linear list, that is often doubly-linked and/or circularly-linked. This means that every block has to have at least one, and maybe two pointers. Besides, in order to support coalescing of adjacent free blocks, boundary tags are usually used which means that each block has both a header and a footer, containing the information on block size and whether it is in use. Several algorithms are possible, depending on the way the linked list is searched and memory allocated. Using *best fit* allocator searches the free list to find the smallest free block large enough to satisfy a request. The strategy is to minimize wasted space. *First fit* searches the list from the beginning and uses the first block large enough to satisfy the request. If the block is larger, it is split and the remainder put into the free list. Several implementations are possible, concerning the order in which the blocks are put back into the list (LIFO, FIFO etc.). *Next fit* uses a roving pointer for allocation. The pointer saves the position where the last search ended, and next search begins from there trying to find the first block that satisfy the request. Usual explanation is faster allocation, but there are some disadvantages concerning locality of objects (objects of different phases of execution are interspersed in memory). Finally, *worst fit* allocates memory from the largest possible block, hoping that the rest of the block will still be large enough for next allocation.

The authors found that in practice best fit and address-ordered first fit (the free blocks in the list are sorted in order of addresses) work well in practice.

2.2.2 Segregated Free Lists

In this type of allocator, an array of free lists is used, and each free list holds blocks of particular size. We mention two variation of this allocation mechanism. *Simple segregated storage* does not allow splitting of large objects in order to satisfy requests for smaller sizes (the opposite case, coalescing, is not allowed either). If no block of that size is available, memory is allocated from the operating system and split in equal blocks of that size (which means that one page contains block of only one size). In the case of *segregated fits* an array of free lists is used, each holding object of the same size class. Size class means that objects of similar sizes are put in the same free list. When memory request is made, its size class is calculated, and any block in the free list of that size class (or bigger) can satisfy the request. Size class schemes usually use sizes of power of 2, but other schemes are used also. If a block of that size class is not available, search is continued in larger size classes. Blocks are split and put into an appropriate free list. They can also be coalesced, using boundary tags, mentioned above.

2.2.3 Buddy Systems

Buddy systems are in fact a special case of segregated fits, using particular splitting and coalescing system. In the simple buddy scheme, the whole heap is split into two large areas, and those areas are further split in the same way, until the appropriate size has been reached. Split blocks are put into free lists according to their size. When blocks are freed, they are merged with its buddy, a unique neighbor in the same level in the binary hierarchical division. The merge is efficient, done by simple address computation. Several variations on buddy systems exist : binary buddies, Fibonacci buddies, weighted buddies, double buddies. All of them are variations on the size of the split block, in order to eliminate as much as possible internal fragmentation.

2.2.4 Bitmapped Fits

In this mechanism a bitmap is used to record which parts of the heap are in use, and which parts are not. A bitmap is an array of one-bit flags, one for each word (or double-word, depending on the architecture) of the heap area. Bitmaps may have an advantage over headers, if the objects size is small (a bit per word incurs a 3% overhead, whereas for a 10-word object, a header incurs 10% overhead).

2.2.5 Conclusion

Memory allocation mechanisms differ mostly by the speed of search for a free block, and by flexibility of splitting and coalescing. It seems that segregated fits using boundary tags, or bitmapped techniques give rather satisfactory results. It is out of the scope of this work to enter into the details of advantages and disadvantages of each mechanism, the purpose of this section was to give just a global overview.

2.3 Basic Garbage Collection Algorithms

In this section we first introduce some base definitions necessary for the understanding of garbage collection algorithms, and later we describe basic and advanced garbage collection techniques.

2.3.1 Phases of Garbage Collection

There are two phases of garbage collection: garbage detection and garbage reclamation. *Garbage detection* consists of distinguishing the live objects from the garbage in some way, while *garbage reclamation* deals with reclaiming garbage objects memory, so that the running program can use that memory space again.

The liveness criterion is defined in terms of reachability from the active variables. These includes statically-allocated global or module variables, local variables in activation records on the activation stack, and any variables currently in registers. These objects are called *root set*. Objects on directed path of pointers from the roots are considered alive; objects not reachable from the root set are considered garbage.

Object representation is done assuming that it should be easy to determine the type of an object at run time. This can be accomplished using either hidden (unavailable to the programmer) *"header" fields* on heap objects or *tagged pointers*. Header field is an extra field containing type information, which is typical for statically typed languages. In order to represent tagged pointers, a shortened representation of the hardware address is used, with a small type-identifying field in place of the missing address bits, which is typical for dynamically typed languages.

The garbage detection phase of garbage collection may be done in two ways: by *reference counting* or directly (maintaining a count of the number of pointers to each object), or by *tracing* or indirectly (traversing the pointers in order to find all the objects the program might reach). There are several varieties of tracing collection (garbage reclamation phase included): *mark-sweep*, *mark-compact*, *copying* and *non-copying implicit reclamation*. We now examine each of these techniques in turn, by first describing it, and then pointing to the associated issues (advantages and limitations).

2.3.2 Reference Counting

Philosophy

Each object has an associated reference count i.e. count of the references (pointers) to it. Every time a reference to the object is created, e.g. when a pointer is copied from one place to another by an assignment, the pointed-to object's count is incremented, and when an existing reference is destroyed, it is decremented. The memory is reclaimed when the object's reference count is 0.

Each object typically has a *header field* of information describing the object, which includes a subfield for the reference count.

When the object is reclaimed, its pointer fields are examined, and any object it points to also has its reference count decremented. Reclaiming one object may therefore lead to reclaiming many other objects.

Algorithms

In this section we present algorithms that are most often used to implement reference counting, starting with simple recursive freeing. The second algorithm, non-recursive freeing, tries to avoid an overhead of the first one. Finally, to optimize the reference counting, a deferred reference counting may be used.

Simple recursive freeing

Description This is the simplest reference counting algorithm. At the beginning, all cells are placed in a pool of free cells (usually implemented as a linked list), `free_list`. The cells are linked by the pointer to the next free cell, `next`. Function *new()* allocates a new cell, `newcell`, from the free list using the function *alocate()*. Function *update(R,S)* updates a pointer from the cell R to the cell S, incrementing the reference count of S. The name of the algorithm comes from the function *delete(T)*, which deletes a pointer to T, decrementing its reference count, and, if its count is 0, it deletes all the pointers from T by calling recursively the function *delete(T)*.

The functions mentioned above are explained hereafter.

Function *new()* Checking for the free memory, calling of the function for the allocation of a new cell, and updating RC (reference count) field of the new cell to 1:

```
new() =
    if free_list == nil
        abort ''Memory exhausted''
    newcell = allocate()
    RC(newcell) = 1
    return newcell
```

Function *allocate()* Allocation of a new cell from the list of free cells:

```
allocate() =
    newcell = free_list
    free_list = next(free_list)
    return newcell
```

Function ***update()*** Updating the pointer fields under reference counting. First the old pointer is deleted (call of *delete*), then the RC of the new pointed-to object is incremented and, finally, the pointer is updated.

```
update(R,S) =
    delete(*R)
    RC(S) = RC(S) + 1
    *R = S
```

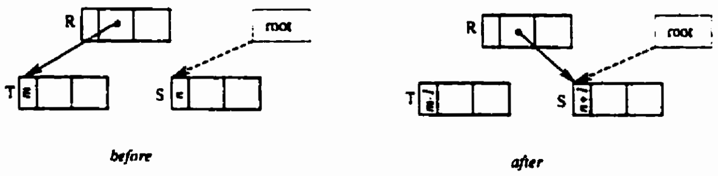

Figure 2.1: `Update(left(R), S)`

Function ***delete()*** Deleting of a pointer to an object T. First the object's RC is decremented, and checked if it is 0. If so, the pointers to the object's children are deleted and the object is freed (call of `free`).

```
delete(T) =
    RC(T) = RC(T) - 1
    if RC(T) == 0
        for U in Children(T)
              delete(*U)
        free(T)
```

Function ***free()*** Putting the garbage object in the list of free objects:

```
free(N) =
    next(N) = free_list
    free_list = N
```

Important Issues This algorithm has many advantages. It is simple to implement, it liberates the non-used memory immediately, its overheads are distributed throughout the computation (which makes it suitable for real-time systems). But, one of its disadvantages lays in the overhead: the cost of removing the last pointer is unbounded since any descendants reachable only from that object must also be freed. The *non-recursive freeing* tries to solve this problem.

Non-recursive freeing

Description Simple recursive freeing distributes processing overheads unevenly: the cost of deleting the last pointer to an object is not constant, it depends on the size of the sub-graph rooted at that object. To avoid that, when the last pointer to a node N is deleted by the function *delete(N)*, N is simply pushed onto a free-stack. And when N is about to be reallocated from the top of the free-stack, any pointers in N are deleted by *new()*, and any immediate referent which would have a RC of zero is pushed back onto the free-stack. The reference-count field is used to chain the free stack.

Function *new()* Checking for the free memory, allocating of a new cell by the function *allocate()* and deleting every child of the new cell by the function *delete*.

```
new() =
    if free_stack == nil
        abort ''Memory exhausted''
    newcell = allocate()
    for N in Children(newcell)
          delete(*N)
    RC(newcell) = 1
    return newcell
```

Function *delete(N)* Checking for the N's reference count: if it is equal to 1, then put N on the free stack, else decrement its RC.

```
delete(N) =
    if RC(N) == 1
        RC(N) = free_stack        — RC field used to chain the free stack
        free_stack = N
    else RC(N) = RC(N) - 1
```

Important Issues This algorithm preserves the advantages of the first one, and it gives a solution for the overhead of updating the reference counts when deleting the last pointer to an object. But it leaves some problems unsolved. One of them is the efficiency problem. The cost of the reference counting is proportional to the amount of work done by the running program, with a fairly large constant of proportionality. When a pointer is created or destroyed, its reference count must be adjusted. In the case of short-lived stack variables, reference counts are incremented and then decremented back to their original value very soon. It is desirable to optimize it to avoid such an overhead. One of the ways to optimize it is to use *deferred reference counting*, which we detail in the next section.

Deferred reference counting

Description Rather than always adjusting reference counts and reclaiming objects whose counts become zero, references from the local variables are not included in this book-keeping most of the time. From time to time, the reference counts are brought up to date by scanning the stack for pointers to heap objects. The cost is still roughly proportional to the amount of work done by the running program, but with lower constant of proportionality.

One of the algorithms for deferred reference counting is the Deutsch-Bobrow algorithm [6]. Here reference counts only reflect the number of references from other heap objects: references from the stack are not counted. This means that objects can no longer be reclaimed as soon as their reference count drops to zero since they might still be directly reachable from a local or temporary variable. Instead, cells with a reference count of zero are added to a *zero count* table (ZCT) by the function *delete*. Function *update(R,S)* updates the pointer from the object R to the object S, and the function *reconcile()* checks whether there are objects in the ZCT that are not present in the stack, and returns them to the free list.

Function *delete(N)* This function decrements RC of the cell to be deleted, checks if RC is zero and in that case puts the cell to the zero count table.

```
delete(N) =
    decrementRC(N)
    if RC(N) == 0
        add N to ZCT
```

Function *update(R,S)* Entries in the ZCT are deleted and the RC incremented when a reference to the object is stored in another heap object.

```
update(R,S) =
    delete(*R)
    incrementRC(S)
    remove S from ZCT
    *R = S
```

Function *reconcile()* Periodically the ZCT is reconciled to remove and collect garbage. Any object with a reference in the ZCT that is not also found in the stack must be garbage and can be returned to the free-list. First the RC of the stack objects is incremented in order to mark the stack. After that, the RC of every object in ZCT is checked. If it is equal to zero, it means that the object is not in the stack, and it is therefore freed.

```
reconcile() =
    for N in stack                          — mark the stack
        incrementRC(N)
    for N in ZCT                            —reclaim garbage
        if RC(N) == 0
            for M in Children(N)
                delete(*M)
            free(N)
    for N in stack                          —unmark the stack
        decrementRC(N)
```

Important issues Deferred reference counts reduces the cost of pointer writes. On the other hand, there is a space cost of ZCT and lack of immediate recycling of the memory (as the garbage is retained until the ZCT is reconciled). There is also a question of stack overflow: ZCT is reconciled when overflowed, but freeing of an object can push more objects to the ZCT, causing the overflow again. This problem can be remedied by either canceling the freeing of the object until the next reconciliation, or implementing the stack as a bitmap (a bit for every word in the heap; an object is entered or removed from the bitmap by setting its bit).

Other reference count techniques

In order to save space required to store reference counts, which can theoretically be large enough to hold the total number of pointers in the heap and in the roots, a *limited-field reference counting* can be used. Small reference count fields may overflow, and precautions must be taken to avoid it. Besides, when the reference count reaches its maximum, it cannot be reduced, because the true count may be greater than its reference count. A backup tracing collector must be used to restore true reference counts. The use of a tracing collector is not burdensome since it is likely to be used to collect cyclic garbage (see Disadvantages below). There is a more radical modification of this technique, which uses a single bit to reference counts. The bit is used to distinguish the pointers that are unique from the shared ones. As in the previous case, the use of the tracing collector is necessary: shared cells can only be reclaimed by tracing.

The execution time of the reference counting is generally greater than that of tracing techniques. In order to profit from the benefits of the reference counting and still have an acceptable execution time, *hardware support* must be included. There is some work in that field (such as self-managing heap memories based on reference counting), but we will not detail it here, because it is not widely used.

Some researchers tried to overcome the main disadvantage of the reference counting, the cyclic garbage (see Disadvantages below), by either treating a cycle as a single entity (under some restrictions), or by distinguishing the pointers internal to the cycle from external references. None of these schemes have been adopted for use by significant systems.

Advantages and disadvantages of reference counting

Advantages This system can perform with little degradation when almost all of the heap space is occupied by live objects. It is useful for finalization (clean-up actions, like closing files, when objects die). As for the usage, this system is not convenient for high-performance implementation of general-purpose programming languages. However, it is used by most file systems to manage files/disk blocks and in simple interpretive languages and graphical toolkits.

Disadvantages Besides certain amount of optimization, there is a cost of book-keeping of the objects whose count is 0 (typically, one or more lists of reusable objects is created by linking the freed objects). Reclamation operation costs few tens of instructions per objects, which is proportional to the number of objects allocated by the running program. There is still an incapability of reclaiming circular structures. If the pointer in a group of objects creates a (directed) cycle, the objects' reference counts are never reduced to zero, even if there is no path to the objects from the root set (in the Fig. 2.3.2 after `delete(right(R))` the cycle STU is neither reachable nor reclaimable). That means that some other kind of garbage collector has to be included, which can compromise the real-time nature of the algorithm.

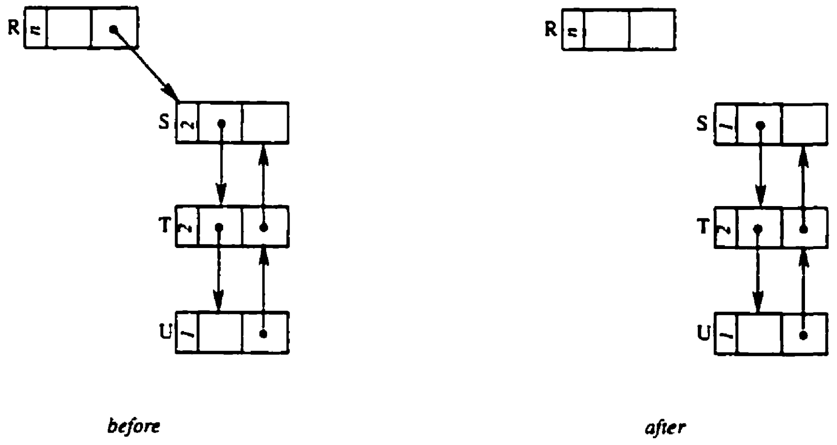

Figure 2.2: Reference counting cyclic data structures

2.3.3 Mark-Sweep Collection

Philosophy

As we mentioned in Section 2.3.1, the first tracing technique for garbage collection is *mark-sweep collection*. The name comes from two phases of garbage collection : garbage detection phase called *mark phase*, and garbage reclamation phase called *sweep phase*. Mark phase consists of tracing live objects by starting at the root set and actually traversing the graph of pointer relationships by either depth-first or breadth-first traversal (see Fig. 2.3.3 : all unmarked cells (with unshaded mark-bits) are garbage). Reached objects are *marked* by toggling a bit in the header of the object or by recording them in a bitmap or in a table. Once the live objects have been marked, memory is *swept* (exhaustively examined) to find all unmarked objects which are then linked into one or more free lists.

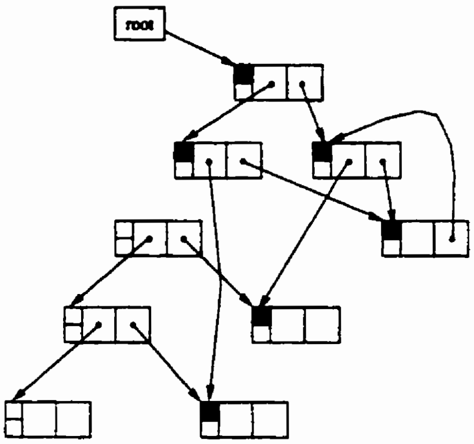

Figure 2.3: The graph after the marking phase

Algorithms

The common characteristic of all mark-sweep algorithms is that a bit associated with each cell is reserved for marking. To mark all the objects that are alive, the function *mark* is called on every member of the root set. After that the unmarked cells are returned to the free pool by the function *sweep*.

```
mark_sweep() =
    for R in Roots
          mark(R)
    sweep()
    if free_pool is empty
          return ''Memory exhausted''
```

Marking phase can be done in several ways, while not many algorithms exist for the sweeping phase. We first mention the marking algorithms, namely: *simple recursive marking*, *using a marking stack*, *using a pointer reversal* and *using a bitmap marking*. After that, we present briefly a simple algorithm for the sweeping phase, and its modification, *lazy sweeping*.

Simple recursive marking The simplest and the least efficient marking algorithm is simple recursive marking.

Description Function *mark(N)* first checks if N is marked; if not, it marks it and calls recursively the function *mark(N)* for every child of N.

```
mark(N) =
    if mark_bit(N) == unmarked
        mark_bit(N) = marked
        for M in Children(N)
            mark(*M)
```

Important issues The problem is the recursion which is neither time- nor space-efficient, and may cause the system stack to overflow. The "using of a marking stack" approach, explained below, makes space and time cost of the marking phase explicit (the maximum size of the marking stack depends on the size of the longest path that has to be traced through the graph; overflows in real time are rather rare).

Using a marking stack

Description A standard method for improving the performance of recursively described algorithms is to replace recursive calls by iterative loops and auxiliary data structures. In the case of marking, an auxiliary stack can be used to hold pointers to nodes that are known to be live but have not yet been visited.

In the following algorithm the method traverses each *node*, stacking branch points only once:

```
gc() =
     mark_heap()
     sweep()

mark_heap() =
    mark_stack = empty
    for R in Roots
        mark_bit(R) = marked
        push(R, mark_stack)
        mark()

mark() =
    while mark_stack ≠ empty
       N = pop(mark_stack)
       for M in Children(N)
           if mark_bit(*M) == unmarked
               mark_bit(*M) = marked
               if not atom(*M)
                   push(*M, mark_stack)
```

In the alternative algorithm, the method traverses each *arc*. General directed graph usually contains more arcs than nodes, hence this algorithm is less efficient:

```
mark() =
    while mark_stack ≠ empty
        N = pop(mark_stack)
        if mark_bit(*M) == unmarked
            mark_bit(*M) = marked
            for M in Children(N)
                push(*M, mark_stack)
```

A graph may contain large nodes, which is not a problem if the nodes are atomic, but if they are not, pushing all its children can cause the stack to overflow. One solution is to put in the marking stack two pointers to the start and the end of each object pushed onto the stack. At each iteration of the marking loop these pointers are examined. If the object is small, then all of its children are pushed onto the stack, otherwise only a portion is pushed and the two pointers are updated with the start and the end of the rest of the object.

Important issues Attention must be paid to the stack overflow. It can be detected in two ways: either by an in-line check in each push operation or by counting the number of pointers contained in the node popped from the stack at each iteration of the marking loop. An alternative is to use a write-protected page (*guard page*): the last page on the stack is set to be write-protected. Memory protection fault is triggered if a stack entry is pushed onto this page. This operation is quite expensive and not much in use. The other inconvenience is the additional space for the stack. Below we explain a method for marking in constant space: pointer reversal.

Using a pointer reversal

Description Previous algorithm needs the use of an extra space for the stack. Pointer reversal is a method of marking in constant space, without using additional space for marking stack. In order to record all branch points that it passes through, the marking algorithm has to store the back pointer to the previously marked node. One way to do that is to use *a pointer reversal* algorithm developed by Deutsch, Schorr and White [7]. This algorithm supposes that all branch-nodes are binary (with exactly two pointer fields: *left* and *right*).

Three variables are used: `current` — the current node, `previous` — the node behind the current node, and `next` — ahead the current. Initially `current` is set to the root of the graph to be marked and `previous` to nil. There are three phases in this algorithm. In the first phase, a function called `mark` follows `left` pointers and marks nodes until it reaches a marked node or an atom. When it arrives there (second phase), it sets a flag-bit of the `previous` node (that indicates that its left subgraph is marked) and attempts to start marking from the `right` node. The original value of the `left` field is restored. In the third phase, the original value of the `right` field is restored and the algorithm retreats to the parent node. This phase is repeated until a node whose flag-bit is not set is found (its right subgraph is not yet marked). The algorithm terminates when `previous` becomes `nil` again.

```
mark(R) =
    done = false
    current = R
    previous = nil
    while not done
        — follow left pointers
        while current ≠ nil
        and mark_bit(current) == unmarked
            mark_bit(current) = marked
            if not atom(current)
                 next = left(current)
                 left(current) = previous
                 previous = current
                 current = next

        —retrait
        while previous ≠ nil
        and flag_bit(previous) == set
            flag_bit(previous) = unset
            next = right(previous)
            right(previous) = current
            current = previous
            previous = next
        if previous == nil
            done = true
        else
            —switch to right subgraph
            flag_bit(previous) = set
            next = left(previous)
            left(previous) = current
            current = right(previous)
            right(previous) = next
```

Pointer reversal can be used for variable-sized nodes. Each node has two additional fields: one holds the number of pointers (`n`) contained in the node (which is necessary in any case) and the second is used for marking (`i`). Each time a child is marked, `i` is incremented. When `i` becomes equal to `n`, the algorithm retreats to the parent node.

Important issues Pointer-reversal algorithms require constant space to operate, but they involve an overhead in each node of the heap. Their performance is consid-

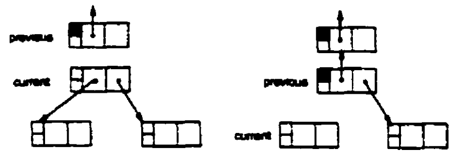

Figure 2.4: The advance phase

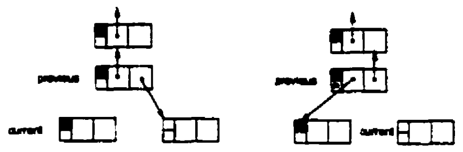

Figure 2.5: The switch phase

erably worse than that of "the pointer-stack method" one, having to visit each node more times than stack-based marking. It should be used as a method of last resort, invoked only on pointer-stack overflow.

Using a bitmap marking

Description The mark bits are not placed in the objects they mark. They can be stored in a separate bitmap table. A bit in the table is associated with each address in the heap that may contain an object. The maximum fraction of the heap occupied by the table is inversely proportional to the size of the smallest object (bigger the object, lesser the number of objects that can fit in the heap, and smaller the bitmap table).

Important issues Bitmaps minimize the amount of memory needed to store mark information. If it is small it can be held in RAM (so that reading/writing of mark-bits does not cause page faults), and 32 bits can be checked at once by ALU operation. During the marking phase, the heap objects are not written into, and atomic objects are not even touched by the collector. The only disadvantage of the bitmaps is that mapping the address of an object in the heap to a mark-bit is more expensive than it would be if the mark were stored in the object.

Lazy sweeping

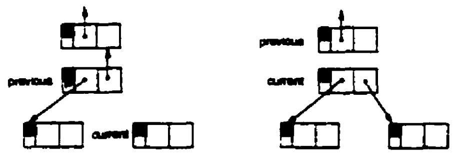

Figure 2.6: The retreat phase

Description One of the disadvantages of mark-sweep garbage collection is that its cost depends on the size of the heap, because the sweep phase must examine the whole heap. The pauses can be reduced if the sweep phase is done in parallel with program execution. The simplest way to do this is to execute a fixed amount of sweeping at each allocation. Hughes's algorithm [8] is an example of such a lazy sweeping. At each allocation, the heap is swept and the memory returned by sweeping is used for the allocation. There is no use of free lists.

```
allocate() =
    while sweep < Heap_top
        if mark_bit(sweep) == marked
            mark_bit(sweep) = unmarked
            sweep = sweep + size(sweep)
        else
            result = sweep
            sweep = sweep + size(sweep)
            return result
    — heap is full
    mark_heap()
```

Lack of free list manipulation is an advantage if mark bits are stored in the objects themselves. But in the case of a bitmap, there is no advantage in reloading and saving bitmap indexes and bitmasks at each call to `allocate`. It is better to either use a free-list (the case of Boehm-Demers-Weiser conservative collector, see Section 2.6.2) or a fixed-size vector (Zorn's generational mark-sweep collector. see Section 2.4.2).

Important issues The cost of mark-sweep collection is likely to be dominated by marking and not by sweeping phase. Yet, there is no reason to sweep the entire heap, while the sweep phase can be done in parallel with program execution by lazy sweeping.

Selective sweeping Chung [9] developed another algorithm that can improve the efficiency of the sweep phase : selective sweeping. We explain it in details in the section 4.3.2, when describing its actual implementation.

This algorithm avoids having to touch every object in the heap during the sweep phase by constructing a set of live objects during the marking phase. Objects in this set are then sorted by address and the gap between each two objects freed in a constant time.

Important issues The efficiency of the selective sweeping depends on the number of live objects. If this number is small, selective sweeping obviously reduces sweeping time. If it is rather big, then traditional sweeping may be better (avoiding the sorting phase, for instance). In order to decide which algorithm to chose, the number of live objects is tracked during marking phase, and if that number exceeds certain threshold, the set of live objects is not constructed and the traditional sweeping is performed.

Advantages and disadvantages of mark-sweep collection

The advantage of this technique over reference counting is that cycles are handled naturally and there is no overhead in pointer manipulations. On the other hand, there is a problem of fragmentation that is not unique for this type of collection. It is difficult to handle objects of varying sizes without fragmentation of available memory. This can be mitigated somewhat by separate lists for objects of varying sizes, and merging adjacent free spaces together (but difficulties remain). The cost is proportional to the size of the heap, including both live and garbage objects, but if live objects tend to survive in clusters of memory, this can greatly diminish the constant of proportionality. Objects of very different ages are interleaved in memory (unsuitable for most virtual memory applications).

2.3.4 Mark-Compact Collection

The disadvantages of mark-sweep collection are reduced by mark-compact collection.

Philosophy

Mark-compact collection remedies the fragmentation and allocation problems of mark-sweep collection. It has three phases : *marking*, *compacting* and *updating the pointers*. Marking is done in the same way as for mark-sweep (see Section 2.3.3). After marking, live objects are slid to one side of the heap adjacent one to another, which creates a single contiguous free space at the end of the heap. Finally, the values of the pointers that referred to moved objects are updated.

Algorithms

In this section we present the most frequently used algorithms for this garbage collection technique : *two-finger algorithm*, *the Lisp 2 algorithm* and *table-based methods.*

Two-Finger Algorithm

Description This algorithm is generally applicable only to fixed-size cells. Two pointers to the heap are used : one to point to the next free location (`free`), the other to the next active cell to be moved (`live`). The forwarding address is left in their old location.

First the live data is marked (using the function *mark()*), and the number of live cells is returned.

```
compact_2finger() =
    no_live_cells = mark()          — marking of the live objects
    relocate()                      — relocation of the cells leaving
                                      the forwarding address
    update_pointers(no_live_cells)  — updating the pointers to the newly
                                      relocated cells
    free = no_live_cells + 1        — first free slot on the heap
```

Then the first pass of the algorithm relocates cells from the upper part of the heap to the holes in the lower part of the heap, overwriting the first field of the vacated slots with the forwarding addresses.

```
relocate() =
    free = Heap_bottom
    live = Heap_top
    while marked(free)              — find the first hole
        free = free + 1
    while not marked(live)          — find the first live cell
        live = live - 1
    while live > free
        move(live, free)
        Heap[live] = free           — leave forwarding address
        while marked(free)
            free = free + 1
        while not marked(live)
            live = live - 1
```

The second pass scans the live cells, all of which are now at the bottom part of the heap. This pass updates the values of any pointer fields that refer to cells that have been evacuated (i.e. with addresses greater than nlive), by referring to the forwarding addresses left by the first pass.

```
update_pointers(nlive) =
    for i = 1 to nlive
        for j in Children(Heap[i])
            if Heap[j] > nlive      — points to relocated area
                Heap[j] = Heap[Heap[j]]
```

Important issues This algorithm can be easily extended to deal with the case when variable-sized cells are allocated in different regions of the heap. Here the mark phase must calculate `no_live_cells` for each region and the cells in each region must be relocated separately.

The chief drawback of this algorithm is that the order in which the cells are relocated is arbitrary, so it is not suitable if the reason for compaction is to improve spatial locality. The Lisp 2 algorithm, presented in the next paragraph, outcomes this drawback, by preserving the cell order.

The Lisp 2 Algorithm

Description This algorithm first marks all live cells (function *mark()*). After the marking, it consists of three phases (explained below) : first it computes the forwarding addresses of each cell and writes them in a special field of each cell (function *compute_addresses()*), then it updates the pointers to the cells to be relocated (function *update_pointers()*) and finally relocates the cells to their new addresses (function *relocate()*).

```
Compact_LISP2() =
    mark()
    compute_addresses()
    update_pointers()
    relocate()
```

The first phase computes the new address of each active cell, and puts it in the `forwarding_address` field of each object. If the object is not marked (i.e. if its `forwarding_address` field is nil), the function *combine()* finds the next live object.

```
compute_addresses() =
    free = Heap_bottom
    P = Heap_top
    while P ≤ Heap_top
        if forwarding_address(P) ≠ nil
            forwarding_address(P) = free
            free = free + size(P)
        else combine(P)
        P = P + size(P)

combine(P) =
    — P is unmarked
    next = P + size(P)
    while forwarding_address(next) == nil          —not marked
        size(P) = size(P) + size(next)
        next = P + size(P)
```

The second pass updates the values of pointer fields of active cells (including root pointers).

```
update_pointers() =
    for R in Roots
        R = forwarding_address(R)
    P = Heap_bottom
    while P ≤ Heap_top
        if forwarding_address(P) ≠ nil
            for Q in Children(P)
                Heap[Q] = forwarding_address(Heap[Q])
        P = P + size(P)
```

The third pass clears the forwarding_address field and moves cells to their new address. At the end of this phase, all active data are compacted to the lower part of the heap, and free indexes the first free location in the heap.

```
relocate() =
    P = Heap_bottom
    while P ≤ Heap_top
        temp = P + size(P)
        if forwarding_address(P) ≠ nil
            free = forwarding_address(P)
            forwarding_address(P) = nil
            move(P, free)
        P = temp
    free = free + size(free)
```

Important issues This algorithm is suitable for cells of variable sizes, it preserves their order, which improves spatial locality. On the other hand, it makes three passes instead of two, and it needs one more pointer-sized field that serves for storing the forwarding addresses and for marking process. The table-based methods, presented below, preserve cell ordering without any space cost.

Table-based methods

Description These methods use *break table* to keep account of the location of blocks of active data and the size of holes, and use this information for updating pointers.

```
compact_table() =
    nlive = mark()
    relocate()
    sort_table()
    update_pointers(nlive)
```

After marking the active graph, a break-table of relocation information is constructed in the free area. The break table is built as each contiguous area of active data is compacted by determining the address of the start of the area (a_i), and the total amount of free space discovered so far (s_i). The pair (a_i, s_i) is written into the free slot at the end of the break table. As areas of active data are relocated toward the bottom of the heap, it may be necessary to move the break table in the opposite direction. If this movement causes the break table to be unsorted, the table must be sorted. Finally, the pointer fields have to be adjusted. To adjust a pointer p, the break table is searched for adjacent pairs (a,s) and (a', s') such that $a \leq p < a'$. The adjusted value of p will then be $p - s$.

Important issues Table-based methods make two passes of the heap and require no extra space (the information is stored in holes themselves). The main concern is the time needed for searching of the break-table which depends on its size. The search can be improved by using a hash table.

Advantages and disadvantages of mark-compact collection

The elimination of fragmentation problems by compacting reduce the cost of allocation, allowing an easy allocation of objects of various sizes. Mark-compact collection preserves the order of objects in memory, which ameliorates spatial locality.

The execution of the compactor can be rather slow. At least two (and in one case, three) passes over the data are required (like in case of mark-sweep). Mark-compact collection can be significantly slower than mark-sweep if a large percentage of data survives to be compacted.

2.3.5 Copying Garbage Collection

Philosophy

Copying garbage collection is another kind of tracing collection. This kind of garbage collection is similar to mark-compact, but without marking: traversal of data and the copying process are integrated, so that most objects need to be traversed only once. Like with mark-compact collection, there is no real collection of garbage. The usual term for the copying traversal is *scavenging*, since only the worthwhile objects amid the garbage are saved.

Algorithms

A Simple Copying Collector

Description The most used copying collector is semispace collector that implements Cheney's algorithm ("stop-and-copy") [10]. The heap is divided into two contiguous semispaces. During normal execution only one of them is used (the allocation is simple and fast because of the large, contiguous free space). When the program demands an allocation that will not fit in the unused area, the program is stopped and the copying garbage collector is called to reclaim space. All the live data are copied from the current semispace (*fromspace*) to the other semispace (*tospace*). After that, the tospace is made "current" semispace, and the execution is resumed. The roles of two spaces are reversed each time the garbage collector is invoked.

The simplest algorithm for copying is Cheney's algorithm [10]. The immediately reachable objects form the initial queue of objects for a breadth-first traversal. Each time a pointer into fromspace is encountered, the referred-to object is transferred to the end of the queue, and the pointer to the object is updated to refer to the new copy. The free pointer is then advanced and the scan continues. Eventually, the scan reaches the end of the queue, signifying that all reached (and copied) objects have also been scanned for descendants.

```
flip() =
    Fromspace, Tospace = Tospace, Fromspace
    scan = free = Tospace
    for R in Roots
         R = copy(R)
    while scan < free
        for P in Children(scan)
    *P = copy(*P)
    scan = scan + size(scan)

copy(P) =
    if forwarded(P)
        return forwarding_address(P)
    else
        addr = free
        move(P, free)
        free = free + size(P)
        forwarding_address(P) = addr
        return addr
```

In order to assure that the objects reached by multiple paths are not copied to tospace multiple times, a slightly more complex process is needed. When the object

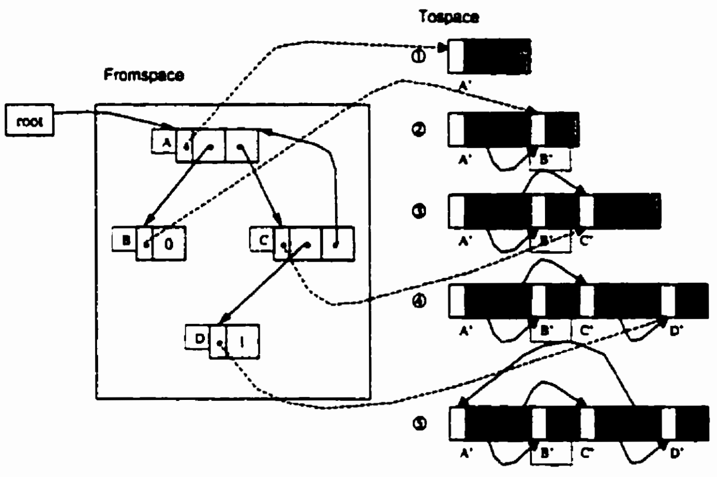

Figure 2.7: Copying of the list [0,1,0,1, ...]

is copied, a forwarding pointer is installed in the old version. It indicates where to find a new copy of the object. Figure 2.7 shows the copying of the list [0,1,0,1, ...]. Note that the forwarding addresses are shown only at the phase in which the object is copied (their presence should be assumed from then on). The copying proceeds in five phases, marked by numbers in the Fig. 2.7 : 1. the root node (A) is copied; 2. space is reserved for B, and then B is copied completely; 3. C is copied; 4. left(C) is copied, space is reserved for D, and then D is copied and 5. right(C) is copied, and the collection is complete.

Important issues The work done at each collection is proportional to the amount of live data at the time of garbage collection, and not to the entire heap. If approximately the same amount of data is live at any given time during the program's execution, decreasing the frequency of garbage collections will decrease the total effort. To do that, the amount of memory in the heap can be increased: program will run longer before filling it, and the average age of objects at garbage collection time will be increased, so the chance that an object will never have to be copied is increased. Here the paging costs are ignored (they can make the use of a larger heap area impractical if there is not correspondingly large amount of RAM).

Advantages and disadvantages of copying collection

Copying collection divides the heap into two semi-spaces, but it uses no further heap memory (mark-bits are not required, and forwarding addresses can usually be written over user data fields). The cost of the collection depends on the amount of live data either than the entire heap.

One of the disadvantages of the copying collection is the copying of large objects, which is more expensive than their marking. This is the reason for having separate large-object spaces in some collectors.

2.3.6 Non-Copying Implicit Collection

Philosophy

In the copying collector, the spaces are a particular implementation of sets. Tracing process removes live objects from one set, and everything that rests is a garbage and can be reclaimed. Given a pointer to an object it must be easy to determine which set it is member of, and it must be easy to switch the roles of the two sets (*toset* and *fromset*). The sets can be implemented as linked lists, and the objects "moved" from one set to another not by copying but by unlinking from one list and linking to another.

Algorithm

Description In order to implement sets as linked lists, this system adds two pointer fields and a "color" field to each object. Pointer fields are for doubly-linked list (set), and a color field indicates which set an object belongs to.

Initially, chunks of free space are linked in one list, and chunks holding allocated objects are linked together into another list. When the free list is exhausted, the collector traverses the live objects and "moves" them from the allocated set to the other one. In fact, the object is unlinked from the `fromset` list, its color field is changed and it is linked into the `toset`'s doubly-linked list.

Important issues The operation of this collector is simple and similar to that of the copying collector. This scheme can be optimized by making the allocation faster: allocated and free lists can be made contiguous and separated only by an allocation pointer. In that case, instead of unlinking the object, allocator can simply advance the allocation pointer.

Advantages and disadvantages of non-copying implicit collection

Tracing cost for large objects is not as high as for the copying collection; the whole object need not be copied (as with mark-sweep). It does not require the actual language-level pointers between objects to be changed (there are fewer constraints on compilers).

The cost is proportional to the number of live objects. The garbage objects are all reclaimed in small constant time. The space costs are comparable to those of a copying collector. There are additional two pointer fields per object, but on the other side no space for both fromspace and tospace version is needed. In some cases fragmentation costs (due to the inability to compact data) may outweigh those savings.

2.3.7 Choosing Among Basic Tracing Techniques

Cost is similar All basic algorithms have roughly similar costs. Criterion for high-performance garbage collection is that its cost is comparable to the cost of allocating objects. Basic cost components of tracing collection are:

- the initial work, such as root set scanning, which is proportional to the size of the root set;

- the work done at allocation, which is proportional to the number of objects allocated, plus an initialization cost proportional to their sizes;
- tracing (garbage detection), which is proportional to the amount of live data.

The latter two costs are usually similar, the third one is usually some significant percentage of the second. That means that the algorithms whose cost is proportional to the amount of allocation (mark-sweep) may be competitive with those having a cost proportional to the amount of live data traced (copying). Currently copying collectors appear to be more efficient than current mark-sweep.

Nonmoving vs. moving collectors In systems where memory is not much larger than the expected amount of live data nonmoving collectors have the advantage of not needing two spaces. Reference counting collectors are also attractive in that case, because their performance is independent of ratio live data/total memory.

Nonmoving collectors can be made conservative with respect to data values that may or may not be pointers. That is useful in the case of the languages like C, and it simplifies the interfaces between modules written in different languages and compiled using different compilers.

Real high-performance systems use hybrid techniques to adjust trade-offs for different categories of objects. Many copying collectors use a separate large object area to avoid copying large objects from space to space. Others use noncopying techniques most of the time, but occasionally compact some data (using copying techniques) to avoid fragmentation.

2.3.8 Important Issues

Every garbage collection technique raises questions concerning the use of memory, the locality of reference, the time efficiency and the conservatism.

Memory If we consider only the copying cost, we could say that it approaches zero as memory becomes very large. On the other hand, large amounts of memory are too expensive. Besides, the poor locality of the allocation and reclamation cycle will generally cause excessive paging. Therefore, it doesn't really pay to make the heap area larger than the available main memory.

Locality The principle of locality has two components: *temporal* and *spatial* locality. Temporal locality means that if a location X is accessed, then it is likely to be accessed again in the near future. Spatial locality means that if location X is accessed, other locations close to X are likely to be accessed in the near future.

The problem is not with the locality of compacted data or with the locality of the garbage collection process itself. Large amounts of memory are touched between the collections. So, the problem is an indirect result of the use of garbage collection: by the time space is reclaimed and reused, it is likely to have been paged out, because too many other pages have been allocated in between the collections. The only way to have good locality is to ensure that memory is large enough to hold the regularly-reused area (see generational collectors).

Time Temporal distribution of tracing can be also troublesome: it can be disruptive for user to have the system become unresponsive for some time while garbage collecting. Generational collectors alleviate this problem, because most garbage collections only operate on a subset of memory.

Conservatism The art of efficient garbage collector design is largely one of introducing small degrees of conservatism which significantly reduce the work done in detecting garbage. The first conservative assumption most collectors make is that any variable in the stack, globals or register, is live even though the variable may actually never be referenced again. Tracing collectors introduce a major *temporal* form of conservatism, simply by allowing garbage to go uncollected between collection cycles. Reference counting collectors are conservative *topologically*, failing to distinguish between different paths that share an edge in the graph of pointer relationships.

2.4 Generational Garbage Collection

2.4.1 Description

Garbage collection techniques introduced in the previous section were the basis for many improvements. One of the most used techniques developed from the basic ones is the generational garbage collection.

Tracing techniques can be improved in several ways. Simple tracing collectors cause delays that can be obtrusive, and its locality of reference, which is important for cache behavior, can be rather poor. Long lived objects are a burden to tracing algorithms because they are repeatedly copied, or marked. On the other hand, researchers found that most objects die young (this hypothesis is known as *weak generational hypothesis*) and therefore storage reclamation is more efficient (in time and locality) by concentrating effort on reclaiming young objects.

In this kind of garbage collection objects are segregated into two or more regions (generations). Different generations can be collected at different frequencies with the youngest generations being collected more frequently. Number of generations varies between implementations. Generational garbage collection is in widespread use in: Lisp, Modula-3, Standard ML of New Jersey, Smalltalk from Apple etc., but whether generational garbage collection is effective or not is application dependent.

2.4.2 Detailed Strategy

Object lifetimes

The age of an object can be measured in two ways: either by wall-clock time or by bytes of the heap allocated. The first one is machine-dependent, it depends on the speed of particular machines and of particular implementations. The count of bytes of heap allocated is a better measure, because it is machine independent and it better reflects demands on the memory management, but it is not a perfect one. Virtual memory algorithms may consider time in their page eviction policy, and objects supporting human interaction have lifetimes determined by the user's activity. Also, some languages have higher rates of memory consumption. All of this make it difficult to measure the age of an object.

Allocation

Objects are first allocated in the youngest generation, but are promoted into the older one if they survive long enough. The youngest generation is collected more frequently, usually by copying, but also by mark-sweep schemes, so pause times will be comparatively short. CPU cycles are saved by not having to copy the older objects from one semi-space to another, although it is still necessary to scan some older objects for pointers into younger generations.

Root set

The root set consists of registers, stack and inter-generational pointers. The latter can be created in either of two ways: by storing a pointer in an object (which can be trapped by write barrier, described later in Section 2.4.2), or when an object containing pointers is promoted to an older generation (which is easily recognized by garbage collector). The write barrier does not have to record assignments to local variables, because they are part of the root set. Also, if younger generations are collected whenever the older one is, only old-young pointers (which are fairly rare) need to be recorded. That means that younger generations can be collected independently of their elders, but not vice versa. Collection of the youngest generation is called *minor collection*, and it is frequent. Collection of several generations is called *major collection* and it is less frequent.

Pause time

Aims of generational garbage collection are: to reduce overall cost of dealing with long-lived objects and to reduce garbage collection pause times. Pause time is dependent on the amount of data that survives the collection, i.e., the size of the youngest generation. The smaller the generation is, the shorter the pauses will be, but the small generation is filled more rapidly. Consequently, the older generation would fill up too soon, resulting in a major collection with a longer pause time. This will also lead to greater number of inter-generational pointers. Ungar and Jackson [11] argue that fixed-age tenuring policies are too restrictive: if the tenuring threshold (i.e., size of the youngest generation) is made too large, pauses will be long; but if very few objects are scavenged at each minor collection, a fixed-age policy will still promote objects even though there is no need to advance any. One way to resolve the problem of widely varying allocation rates is to invoke the collector when the volume of data allocated since the last garbage collection exceeds an allocation threshold. It is presumed that the size of semi-spaces can be varied dynamically. Ungar and Jackson introduce a dynamic advancement mechanism which has two rules:

1. **Only tenure when it is necessary** If few objects survive a scavenge it is probably not worth advancing them, particularly if the cost of write-barrier is high.
2. **Only tenure as many objects as necessary** If the number of survivors suggests that the maximum acceptable pause time would be exceeded at the next scavenge, the age threshold is set to a value designed to advance the excess data.

Number of generations

The number of generations can be greater than two. The use of multiple generations would allow new objects to be promoted quickly, keeping the youngest generation fairly small. This would, on the other hand, introduce greater complexity and larger number of inter-generational pointers. It is found that very large difference in reclamation rates between very new and slightly older objects are not reflected in subsequent generations, so usually two to three generations are used.

Promotion threshold

What should be the promotion threshold, i.e., the number of minor collections that an object must survive before it is advanced to the next generation ? It is shown that the number of objects that survive two scavenges is much less than the number that survive just one scavenge. Increasing the number of scavenges beyond two reduces the number of survivors only slightly. The promotion threshold could be adjusted dynamically.

Handling inter-generational pointers (write barriers)

As we saw, the inter-generational pointers are part of the root set. Thus, it is necessary to find them and the simplest method is to scan older generation at the collection time. It is found that linear scanning is faster and gives better locality than tracing. This technique is used by conservative collectors. There are more precise methods which can be implemented by either hardware or software.

Software barriers are provided by the compiler's instructions before each read and write. Hardware techniques do not require additional instructions and are advantageous in the presence of uncooperative compilers, but they may require hardware components that are not generally available.

There are several methods for trapping and recording inter-generational pointers: *entry tables, remembered sets, sequential store buffers, page marking* and *card marking.*

Entry tables Each generation has an entry table of references from older generations. Every time a reference from an old object to a young one is created, rather than to point directly to the younger object, a new entry to the table is added. The old object points to that entry, and the entry is a pointer to the young object. If the old object already contained a reference to an item in an entry table, that entry is removed. The advantage of this scheme is that when a younger generation is collected, it is only necessary to scavenge its entry table rather than to search every older generation. The disadvantage is a possibility of multiple references to a single object, so the cost of scanning tables is proportional to the number of store operations rather than to the number of inter-generational pointers.

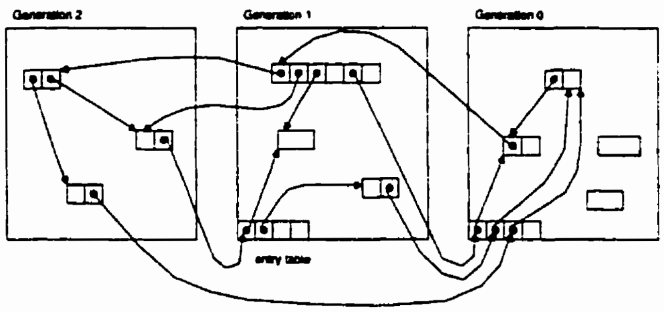

Figure 2.8: Entry tables

Remembered sets In contrast to *entry tables* which record pointed-to objects, *remembered sets* record the old object that contains pointer. In fact, the write barrier intercepts the store to check two things: whether a pointer is to be stored and whether it is a pointer from old to young object. If so, the address of the old object is added to a remembered set. To avoid duplication, each object has a bit in its header indicating whether it is already a member of the remembered set.

The disadvantage of this approach is that if an old object were stored into several times between collections, the checks would be repeated. And if the object were large, then it would have to be scanned in its entirety at the collection time, which has been observed to thrash Tektronix Smalltalk [5].

Sequential store buffers This is a special kind of remembered set, where the write barrier unconditionally adds addresses that might contain pointers to younger generations to the end of the sequential store buffer, and a 'no access' guard page is used to trap overflow. When the buffer is full (when the page is dirtied), then the special routine processes the list, using a fast hash table to remove duplicates. Sequential store buffers are used for Modula-3 and Smalltalk garbage collectors [12].

Page marking Rather than recording which objects have inter-generational pointers stored into them, the virtual page which is stored into is recorded. Page marking can be done with hardware support or with virtual memory support.

Page marking with hardware support was used for Symbolic 3600 machines. Whenever a reference to generational memory was stored in any page, the write-barrier hardware set a bit in the *garbage collector page table* of the corresponding page-frame of physical memory. This methods prevents duplicates, because even if the bit is set many times, the page is going to be scanned only once. As for swapped-out pages, to avoid unnecessary swapping-in, their details are held in the *ephemeral space reference table* (ESRT) which is maintained by software in non-pageable memory. The two tables are searched at collection time, swapping pages only if their bit in ESRT is set. This technique was feasible because of special tag hardware support to make generation checking fast and because pages were fairly small.

Page marking with virtual memory support uses virtual memory *dirty bits* that are used to indicate whether the page has been changed in any way since it was last written out to disk. A copying collector only needs to scan those pages that were written to during or since the last garbage collection. The virtual memory mechanism must therefore be intercepted, which requires modifications of the operating system kernel, or, alternatively, pages may be write-protected by a system call. Reliance on virtual memory protection mechanisms makes this method unsuitable for real-time applications. The problem is also that pages in modern systems are much larger than those of Symbolics 3600, and the cost is increased by recording any modifications to the page (not just inter-generational pointer stores).

Card marking The opposite of page marking would be word marking: when a location is modified a bit in a *Modification Bit Table* is set. The problem is that a bit-table would require a lot of space. The solution is to divide the heap into small regions called cards. The advantage of *card marking* is that the scanning is reduced in comparison to page marking because of smaller size of cards, and the space necessary for a card table is reduced in comparison to word marking. As with word marking, a bit in the card table is set unconditionally whenever a word in a card is modified. To reduce the number of instructions needed, a byte-map, instead of bitmap, can be used. At the collection time, dirty cards are scanned for inter-generational pointers, and if none found, then the dirty bit/byte is cleared. The dirty cards can be gathered onto the same virtual page, the number of pages holding cards to be scanned, and likely to be scanned again, can be reduced.

The most promising write barrier systems For general purpose hardware *sequential store buffers* and *card marking* seem the best. The cost of the write barrier is the same for both (two instructions), but cards' overhead is more predictable, because SSB can overflow. Remembered sets offer precision, but allow duplicates in SSB. One possibility is to use a hybrid, as in Smalltalk interpreter [13]: write barrier uses card marking but older-younger pointers are saved in remembered sets.

Non-copying generational garbage collection

Copy-based generational collectors are conceptually simpler, but it is possible to build mark-sweep based generational collectors. Zorn [14] found that his mark-and-deferred-sweep generational collector performed significantly better, than his copying collectors (Allegro Common Lisp on a Sun 4/280). His collector had four generations, each of which contained a mark bitmap, a fixed-size-object region and a variable sized-object region. The fixed-sized-object region is divided into regions for objects of different size, and is collected by mark-and-deferred-sweep; the variable-sized-object region contains objects that do not fit in any of fixed-sized-object regions and is collected with a two-space copying collector. There is no reason why all generations should be collected in the same way. This is particularly true for the oldest generation. If the copying collection is used, the oldest generation can be divided into two semi-spaces. Or it can be handled by non-copying collector, or may not be collected at all. If a mark-sweep is used, occasional compacting could be done.

2.4.3 Advantages and Disadvantages of Generational Garbage Collection

Generational garbage collection is highly successful in a wide range of applications. Pause times can be reduced to a level where it is worth considering this kind of garbage collection instead of incremental (see Section 2.5). Cache and paging behavior of the application is made better by concentrating allocation and collection to a small region of the heap. The cost can be reduced by delaying collection of long-lived objects. The programs which allocate a large number of short-lived objects and where non-initializing pointer writes are rare benefit the most.

There are some disadvantages of this kind of collection. Short pauses are defeated by large root sets, i.e., unusually high number of global or local variables. If object lifetimes are not sufficiently short, minor collections will reclaim too few objects, which will result in increased promotion, more frequent major collections and bad locality (if only the youngest generation can fit in real memory). The system must be able to distinguish older from younger objects, by using the write barrier, and storing in an old object a pointer to a young one becomes more expensive. Frequent pointer writes increase the overall cost of write barrier.

2.5 Incremental Tracing Collectors

2.5.1 Description

In case of real-time applications garbage collection pauses have to be reduced to satisfy the worst case performance. Hence, small units of garbage collection must be interleaved with small units of program execution. Fine-grained incremental garbage collection appears to be necessary, and one of the techniques that is naturally incremental is reference counting. Unfortunately, reference counting has efficiency and efficacy problems (see Section 2.3.2), and therefore it is desirable to make tracing (copying or marking) collectors incremental.

The main difficulty with the incremental collection is that while the collector is tracing out the graph of reachable data, the graph may change. The running program is therefore called *mutator* which is, from the garbage collector's point of view, simply a concurrent process that modifies data structures that the collector is attempting to traverse. There has to be some way of keeping track of changes related to the graph of reachable objects.

There is therefore a variety of coherence problems: having multiple processes attempt to share changing data, while maintaining some kind of consistent view. An incremental mark-sweep traversal poses a *multiple readers, single writer* coherence problem. Only the mutator writes to pointer fields and only the collector writes to mark fields. Copying collectors pose a more difficult problem: a *multiple readers, multiple writers* problem. Both the mutator and the collector may modify pointer fields and each must be protected from the inconsistencies introduced by the other.

The degree of conservatism is important for this kind of collection, also. The garbage collector's view of the reachability graph is typically not identical to the actual reachability graph visible to the mutator. It is safe, conservative approximation of the true reachability graph. Typically, some garbage objects go unreclaimed for a while, which is unfortunate but necessary in order to avoid very expensive coordination between the mutator and collector. The more we relax the consistency between those two graphs, the more conservative our collection becomes, and the more floating garbage we must accept, but the more flexibility we have in details of the traversal algorithm.

2.5.2 Detailed Strategy

Tricolor marking

Garbage collection algorithms can be described as a process of traversing the graph of reachable objects and coloring them. The objects subject to collection are conceptually colored white, and by the end of the collection the retained objects must be colored black. So, white objects are unreached objects in fromspace (in a mark-sweep collector, objects whose bit is not set); black objects are objects that are moved to tospace (in mark-sweep, objects whose bit is set). To better understand the relationship between the mutator and the collector, a third color is introduced, gray, to signify that objects have been reached, but their descendants may not have been. The traversal proceeds in a wavefront of gray objects, which separates the white (unreached) objects from the black ones. There are no pointers directly from black objects to white ones and if the mutator creates a pointer from a black object to a white one, it must somehow notify the collector.

In order to make garbage collection invalid the mutator has to both: a) write a pointer to a white object into a black object and b) destroy the original pointer before the collector sees it. Read and write barriers are implemented in order to prevent these two events to happen simultaneously.

Coordinating the collector with the mutator

There are two basic approaches: implementing either *read* or *write barrier*. Read barrier detects when the mutator attempts to access a pointer to a white object, and immediately colors the object gray. One of the algorithms that uses a read barrier is

Baker's algorithm [15]. In the case of write barrier, when the program attempts to write a pointer into an object, the write is trapped and recorded. There are two basic methods for write barrier: snapshot-at-beginning and incremental update. Below we give more details about these three read and write barriers methods.

Read barriers

Baker's algorithm The best known real-time garbage collector is Baker's incremental copying scheme. It uses a read barrier for coordination with the mutator. Any fromspace object that is accessed by the mutator must first be copied to tospace. All new objects are allocated at the top end of tospace (they are *black* when allocated), so the new cells cannot be reclaimed in this cycle. In order to ensure that the collector finds all of the live data and copies them to tospace before the free area in new space is exhausted, each time an object is allocated, an increment of scanning and copying is done. In terms of tricolor marking the scanned area of tospace contains black objects, and the copied but unscanned objects are gray. As-yet unscanned objects in fromspace are white. Whenever the mutator reads a (potential) pointer from the heap, it immediately checks to see if it is a pointer into fromspace; if so, the referent is copied to tospace, i.e., its color is changed from white to gray. The read barrier may be implemented in software, by preceding each read (of a potential pointer from the heap) with a check and a conditional call to the copying-and-updating routine. Alternatively, it may be implemented with a specialized hardware checks and/or microcode routines.

The main limitation of Baker's algorithm is that it is closely coupled to mutator, therefore expensive on hardware. The time to access an object depends on whether the object is in tospace or in fromspace, so pauses are inpredictable.

Write barriers

Snapshot-at-beginning To avoid overwriting of pointers without collector's knowledge, at the beginning of garbage collection a copy-on-write virtual copy of the graph of reachable data structures is made. The simplest and best-known snapshot collection algorithm is Yuasa's [16]. If a location is written to, the overwritten value is first saved and pushed on a marking stack for later examination. This way all objects which are live at the beginning of garbage collection will be reached, even if the pointers to them are overwritten. This algorithm is very conservative, it allows the tricolor invariant to be broken, because *all* overwritten pointer values are saved and traversed and no object can be freed during collection. Newly allocated objects are considered black, for collector the reachability graph is a set union of the graph at the beginning of garbage collection plus all of those allocated during tracing.

Incremental Update Incremental update records when pointer to a white object is stored into a black object (either the black or the white object is grayed). There are two similar algorithms: one which is due to Dijkstra et al. [17] and the other due to Steele [18]. Dijkstra'a algorithm attempts to retain the objects that are live at the end of garbage collection. Objects that die during garbage collection — and before being reached by the marking traversal — are not traversed and marked. Precisely, an object will not be reached by the collector if all paths to it are broken at a point that the garbage collector has not yet reached. Objects are allocated white, so at some point the stack must be traversed to preserve the objects reachable at that time. The tricolor invariant is preserved by blackening the pointed-to white object, rather than reverting the stored-into black object to gray. Steele's algorithm, on the other hand, reverts the stored-into black object to gray. It is less conservative than Dijkstra's, because the pointer might be later overwritten, freeing the object. Therefore, it reduces the amount of floating garbage.

2.5.3 Important Issues

The choice of the barrier

The choice depends on relative frequency of reads and writes, how often the barrier is invoked (on every read or just once per page per cycle) and on the amount of work the barrier has to do. Write barriers are usually used for mark-sweep algorithms, and read barriers for copying ones. The cost of write barrier is less than that of the read barrier, without specialized hardware support, a write barrier appears easier to implement efficiently, because heap pointer writes are much less common than pointer traversals.

The amount of floating garbage (conservatism)

As for conservatism, snapshot-at-beginning barrier is the most conservative, and incremental update is less conservative. The amount of floating garbage depends also on the treatment of new cells (black or white allocation). The black allocation is more conservative than white.

Real-time

There are two types of real-time applications. In the case of *hard real-time* applications atomic actions of garbage collector must complete within *guaranteed* time (which is possible only with hardware support). For *soft real-time* applications, atomic garbage collector actions complete within some *reasonable* period of time (all of the presented algorithms satisfy this constraint).

The criterion for real time garbage collection is often stated as *imposing only small and bounded delays on any particular program operation.* The problem is that the notion of "small" delay is inevitably dependent on the nature of an application. Besides, this criterion unrealistically emphasizes the smallest program operations. A more realistic requirement for real time performance is that the application should always be able to use the CPU for a given fraction of the time at a *timescale relevant to the application.*

Some copy collectors use virtual memory protections to trigger pagewise scanning and this may lead to failing to respect real-time guarantees.

Baker's is the best known incremental algorithm, but it may not be suitable for most real-time applications, because its performance is very unpredictable at small timescales. Algorithms with a weaker coupling between the mutator and the collector (such as most write-barrier algorithms) may be more suitable.

Non-copying algorithms have the convenient property that their time overheads are more predictable, their space costs are much more difficult to reason about because of the fragmentation.

An important determinant of real-time performance is the time required to scan the root set. The pauses caused by scanning the root set occur in Baker's incremental collector at the time of a flip, and in incremental update tracing algorithm at the time of termination. One way to bound the work required is to keep the root set small. Some of the local and global variables may be treated like objects on the heap. Reads or writes to these variables will be detected by the read or write barrier, but there is a cost of the barrier to pay.

As the collector tries not to use too much of CPU to meet real-time deadlines, it has a real-time deadline of its own to meet: it must finish its traversal and free up more memory before the currently-free memory is exhausted. In order to achieve this, it is necessary to quantify the worst case, to put some bound on what the collector could be expected to do. The usual strategy is to use an *allocation clock*: each time an object is allocated, a proportional amount of garbage collection work is done. When allocating black new objects do not need to be traversed, and in the worst case the same objects as in a snapshot-at-beginning are retained. The minimum safe tracing rate is proportional to the amount of live data and inversely proportional to the amount of free memory: it approaches zero as memory becomes very large relative to the maximum amount of live data. When allocating white, it is necessary to traverse reachable white objects, and in the worst case we traverse everything we allocate before it becomes garbage. The worst case safe traversal rate therefore approaches the allocation rate.

At the end of the collection, the collector can determine how much live data was in fact traced, and revise downward its worst-case estimate of what could be live in the next collection. Alternatively, if the collector determines that it has less than the worst-case amount of work to do, it may avoid garbage collection activity entirely for a while, then re-activate the collector in time to ensure that it will meet its deadline (in case that read and write barrier can be efficiently disabled).

Choosing an incremental algorithm

The overall average performance and worst-case performance should be considered when choosing an algorithm. Less conservative algorithms may not be more attractive, because they are just as conservative in the worst case. And moreover, they can be *more* conservative in practice, because of their high overhead (costly write barrier) which may keep it from being run as often.

Generational techniques make the overheads of incremental collection unnecessary for many systems where hard real-time response is not necessary. For other systems, it may be desirable to combine incremental and generational techniques, and careful attention should be paid to how they are combined.

2.6 Conservative and Partially Conservative Garbage Collection

2.6.1 Introduction

Languages are usually not implemented with garbage collection in mind, so it is important to know how type accurate garbage collectors are. Three types of collectors can be distinguished, depending on their type accuracy: *type accurate*, *conservative* and *partially accurate, partially conservative.* Type accurate garbage collector can determine unambiguously the layout of any object in registers, the stack, the heap or any other memory area; the collector requires cooperation with the compiler. Conservative garbage collector must assume that every word is a pointer and may not alter the value of any user program data; the collector has no help from the compiler. The third category, partially accurate, partially conservative collector, assumes knowledge of the format of collectible data structures on the heap, but not of the stack layout or of register conventions; the collector requires the programmer or compiler to observe certain conventions for heap allocated data.

A type accurate garbage collector is not possible for languages like C or C++, because they are not made with garbage collection in mind and the collector does not have all the necessary information from the compiler. On the other hand, Boehm-Demers-Weiser collector (conservative) and mostly copying Bartlett's collector (partially accurate, partially conservative) were successfully introduced in these languages. Next sections give more details about these collectors.

2.6.2 Boehm-Demers-Weiser Collector

The best known conservative collector is the *Boehm-Demers-Weiser collector* [19]. This collector is fully conservative and does not rely on any cooperation from the compiler. Values held in data structures used by the user program and its run-time system, including registers ans stack frames may be scanned for potential pointers but are never altered. Therefore the collector must be based on a non-moving algorithm, i.e., on mark-sweep algorithm.

Description

Allocation A program can be thought of using two logically distinct heaps: one maintained by garbage collector and its allocator and one maintained by explicit calls to standard routines (*malloc/free*). Objects in standard heap do not point to objects in collected heap. The heap is made of *blocks* (4Kb), each block containing objects of different sizes. There are separate free lists for each common object's size. Each block has a separate block header held on a linked list.

The heap can be expanded by requesting further blocks. Objects larger than half a block are allocated to their own chunk of blocks; if no free chunk of sufficient size is available the allocator invokes the garbage collector or expands the heap, depending on the amount of the allocation done.

Small objects are allocated by popping the first member of the free list to that size of the object; if the free list is empty the sweep phase is resumed and if no space is reclaimed by the sweep the allocator invokes a garbage collection. If the collection is unsuccessful a new block is obtained from the low-level allocator.

Root and pointer finding Roots can be found in registers, in the stack and in the static areas. The problem is to find these areas, and it is highly system specific. Marking from registers requires assembly code, but its structure is not difficult. For many architectures it consists of pushing the content of a register onto the stack and then calling a C routine to mark from the top of the stack. The next problem is to discover the bottom of the stack and to determine in which direction it grows. It is done either by using explicit knowledge of the run-time system or by taking the address of the first local variable of `main()`. Finally for the static areas the collector is able to handle dynamic link libraries on some systems, in which case the libraries must be re-registered in each collection.

As for pointers, the collector must treat any word it encounters as a pointer unless it can prove otherwise. It has to be able to determine the validity of a pointer accurately and cheaply, with caution not to reclaim a valid data, but without an access in conservativity. An object is only marked as a pointer if it passes each of the three tests:

- Does a potential pointer p refer to the heap ?
- Has the heap block that supposedly contain this object been allocated ?
- Is the offset of the supposed object from the start of its (first) block a multiple of that block's object size ?

If the pointer passes these tests, the corresponding bit in the block header is set, and the object is pushed onto a mark stack (the same as in mark-sweep collection).

Important issues The main problem of conservative garbage collection are *space leaks*, caused by misidentifying data as heap pointers, thereby retaining memory that could otherwise be recycled. The usual case of retaining a large piece of memory are linked lists: if a false pointer points toward an element of the list, all the following elements will also be falsely retained (if the list is to be recycled). There is also a possibility that small integers could be mistaken for pointers: if pointers are not required to be properly aligned, the collector must consider all possible alignments.

The efficiency of Boehm-Demers-Weiser collector was tested on the versions 1.6 and 2.6 and not with the most recent versions. That means that the obtained results do not represent the most optimized collector. Nevertheless, the old version showed a good efficiency (execution time overhead of some 20% above the best of the explicit allocators), altough actual times varied considerably depending on the application program running. The performance of the collector was the best with a program that primarily allocate and deallocate very small objects, but for some programs the overall execution time overhead was up to 57%.

2.6.3 Mostly Copying Bartlett's Collector

One of the collectors that is not fully conservative is *the mostly copying Bartlett's collector* [20]. This algorithm is originally designed to support high level languages that used C as an intermediate language. It still assumes no knowledge of register, stack or static area layouts, but does assume that all pointers in a heap allocated area can be found accurately. Objects that may be referred to from the stack, registers or the static area are treated conservatively and are not moved. Objects only accessible from other heap-allocated objects are copied.

Description The heap is divided on a number of equal-sized *blocks*. Blocks comprising each semi-space do not have to be contiguous: each block contains a space identifier. In order to move an object from one semi-space to another, either the objects can be copied or the identifier of the block changed. Within a block, allocation is done by incrementing a free space pointer. If there is not enough space in a block, the heap is searched for a new free block. Larger objects are allocated over as many blocks as necessary. Garbage collection is initiated when the heap is half full. First the roots are scanned for potential pointers into the heap. The block that contains the pointed-to object is added to tospace by changing the value of the block's space identifier. The block is also appended to the tospace list for scanning. In the next phase all objects in all blocks in tospace are scanned, and each reachable fromspace object is moved into a block in tospace leaving behind a forwarding address. Once tracing is complete, the fromspace identifier is changed to tospace identifier and the garbage collection is complete.

Important issues Mostly copying incurs a small space overhead to store space identifiers, type information and to link the blocks of a space. The collector can be made generational, by using the space identifiers to encode the age of the object. The decrease in performance due to maintaining of remembered sets is compensated by reduction of time spent on garbage collection, at least for larger programs.

2.6.4 Comparison of Two Algorithms

There have not been thorough studies of the efficiency of Bartlet's collector, nor of the comparative performance of the two collectors. The mostly copying collector would perform better in an environment with high allocation rate of short-lived objects (but it is not sure whether it is typical for C). Conservative garbage collection performs well, errors caused by pointer misidentification are unlikely to be an issue.

2.7 Conclusion

This chapter gave an overview of basic memory allocation and garbage collection algorithms. As Java implements garbage collection, we wanted to check which of these numerous algorithms are typically used in Java garbage collectors, and, if possible, to test the efficiency of each such implementation. Next chapter presents garbage collectors in various JVMs, the methodology used to test them and the results obtained.

Chapter 3

Garbage Collection in Various JVMs

3.1 Introduction

It is in fact Java programming language that brought garbage collection into the mainstream, by mandating garbage collection. The efficiency of garbage collection algorithm is one of the main factors that influence the overall efficiency of Java application. In order to improve it, Sun's JDK (Java Development Kit) changed its garbage collection algorithm several times. Many other Java virtual machines have the same or different garbage collectors as Sun's implementations.

This chapter presents the preliminary garbage collector benchmarks made on several JVMs. First, we present the difficulties encountered while choosing a virtual machine (Section 3.2), an application(s) to use (Section 3.3), an operating system and a profiler (Section 3.4). Then, we present and motivate our final choice (Section 3.5). Finally, we present the benchmarks themselves (Section 3.6.2) and conclude.

3.2 Choice of the Java Virtual Machine

First of all, the choice of Java virtual machines to test has to be made. A JVM has to be available, has to have a garbage collection system (preferably well documented), has to be portable if we want to test it on several platforms (which it should be by definition, but that is not always the case) and has to be able to run the applications chosen for profiling.

Here we present the virtual machines found searching exhaustively the Internet. As we are particularly interested in their garbage collection system we briefly present an overall JVM architecture and then pass to a detailed architecture of a garbage collection system. The amount of information depends on its availability : many JVMs are not well documented, or have not a well documented garbage collector. We can separate the found JVMs in two groups : for normal and for embedded systems. Those for embedded systems have either no garbage collector or a not very developed one. Nevertheless, we mention both kinds of JVMs, but we consider for testing just the first one.

3.2.1 Sun's JVM

Sun was the first to develop a JVM, so their JVM is the most robust one. There are several versions of JDK, which come with several JVMs. To our knowledge, there are three types of JVM with different garbage collection systems :

- JDK1.0 - JDK1.1 - JDK1.2.2 (classic)
- JDK1.2.1 (ResearchVM, ExactVM)
- JDK1.2.2 - HotSpot

All of the Sun's virtual machines are freely down-loadable from Sun's site. We will consider the three of them separately.

JDK1.0 - JDK1.1 - JDK1.2.2 (classic)

First versions of Java virtual machine released by Sun had the simplest garbage collection system. The details on its implementation can be found in Java Tutorial [21]. Here we cite the part that explains the mark-sweep algorithm used in these versions of Java virtual machine :

> "... The Java garbage collector is a mark-sweep garbage collector. A mark-sweep garbage collector scans dynamic memory areas for objects and marks those that are referenced. After all possible paths to objects are investigated, unmarked objects (unreferenced objects) are known to be garbage and are collected. A more complete description of Java's garbage collection algorithm might be "a compacting, mark-sweep collector with some conservative scanning."
>
> The garbage collector runs in a low-priority thread and runs either synchronously or asynchronously depending on the situation and the system on which Java is running. It runs synchronously when the system runs out of memory or in response to a request from a Java program.
>
> The Java garbage collector runs asynchronously when the system is idle, but it does so only on systems, such as Windows 95/NT, that allow the Java runtime environment to note when a thread has begun and to interrupt another thread. As soon as another thread becomes active, the garbage collector is asked to get to a consistent state and terminate...."

Java 2 SDK Production Release for Solaris (ExactVM or ResearchVM)

This is the JVM developed by Java Topics group in Sun Labs under the name of ExactVM (lately changed to ResearchVM) and incorporated into Sun's Java 2 SDK

Production Release for Solaris [22]. It uses a generational memory system with two generations [23]. Generation 0 uses a copying collection with two semi-spaces, while generation 1 uses a single mark-compact space. The memory system sometimes allocates very large objects directly in the oldest generation (essentially this happens when they don't fit in the youngest generation). A card table is used as a write barrier.

JDK1.2.2 (HotSpot)

Sun recently released a new virtual machine for Java, named Hotspot, which is claimed to have higher general performance due to handleness (object references are implemented as direct pointers), faster thread synchronization, significantly reduced code space and an accurate garbage collector. Hotspot's garbage collector is possibly coded in C++ because Sun mentions its clean object-oriented design which provides a high-level garbage collection framework that can easily be instrumented, experimented with, or extended to use new collection algorithms.

The Java Hotspot garbage collector is a fully accurate collector, so it can make several strong design guarantees that a conservative collector cannot make. For example, all inaccessible object memory can be reclaimed reliably and all objects can be relocated, allowing object memory compaction, which eliminates object memory fragmentation and increases memory locality.

This collector uses several garbage collection algorithms. First of all, it employs a state-of-the-art generational garbage collector. In the Hotspot's white paper [24] no details are given on the number of generations, but as only the nursery is mentioned, we suppose that there are only two generations. The second generation is probably what is called "old object" memory area. It employs a standard mark-compact collection algorithm, which eliminates memory fragmentation. Supposedly, for the nursery, a copying algorithm is used, as for the ResearchVM (Section 3.2.1). In order to eliminate garbage collection pauses in the second generation, proportional to the amount of live data, an alternative old-space garbage collector is introduced. It is a fully incremental collector based on the "train" algorithm [25]. Since this algorithm is not a hard-real time algorithm, it cannot guarantee an upper limit on pause times; however, in practice much larger pauses are extremely rare, and are not caused directly by large data sets. To our knowledge, the Hotspot and the ResearchVM garbage collectors differ over this incremental collector.

Sun's Hotspot is available for download from Sun's Java Web site [26]. Its source is also available under the Sun Microsystems Community Source Licensing program.

3.2.2 Kaffe

Kaffe [27] is a cleanroom, open source implementation of a Java virtual machine and class libraries. Kaffe mostly complies with JDK 1.1, except for a few missing parts. Some of its parts are already JDK 1.2 (Java 2) compatible.

Despite recent significant improvements, Kaffe does not have a state-of-the-art garbage collector. The current collector is simply a non-incremental, non-generational, conservative mark-sweep collector with a Boehm-like allocator. Despite being non-incremental, it uses linked lists to keep track of objects. As a result, Kaffe spends a larger amount of its execution time than necessary collecting garbage.

What is needed is one or more high-performance collectors for Kaffe. Fortunately, the interface to the garbage collection subsystem is clearly defined, which should allow for independent development of faster collectors.

3.2.3 LaTTe

LaTTe is a Java virtual machine developed starting from Kaffe v. 0.92. It includes a novel JIT compiler targeted to RISC machines (specifically the UltraSPARC). Additionally, the runtime components of LaTTe, including thread synchronization, exception handling, and garbage collection, have been optimized. Like Kaffe, LaTTe also uses a mark-sweep garbage collector which is partially conservative but the internal structure of LaTTe's garbage collector is totally different from Kaffe's. LaTTe runs on Solaris 2.5+ running on UltraSPARCs. Currently, there are no plans to port LaTTe to other architectures. It is freely downloadable from LaTTe's site [2].

3.2.4 Mach J

Mach J [28] is a Java virtual machine developed by Mach J Company. It is written in C++. It does not include a JIT compiler, it relies on its very efficient execution engine. Mach J supports native threads. "Green" threads are not supported or planned. Mach J has a realtime incremental garbage collector. A license fee of $75,000 is required in order to use it.

3.2.5 DynaFlex (TowerJ 3.0)

TowerJ [29] is a multi-platform native Java compiler and runtime environment for process-intensive, server-side Java applications. TowerJ includes new DynaFlex Java virtual machine. It allows application specific performance tuning allowing developers to specify garbage collection, threads, and optimization parameters. No details are given on the garbage collection algorithm. Licence fee for TowerJ is $5,000.

3.2.6 Hewlett-Packard's JVM

Hewlett-Packard (HP) developed two Java virtual machines : one, HP-UX VM v.1.1, is for normal (not memory-constrained) and the other, ChaiVM, is for embedded systems.

HP-UX v.1.1

The first one, HP-UX Virtual Machine with JIT for Java, v. 1.1., was made for Java 1.1. Later HP licensed Hotspot (see Section 3.2.1), which is used as HP JVM for Java 1.2.

HP-UX v.1.1.'s release notes [30] states that this virtual machine has an improved garbage collection by automatically discarding unused classes. No details on the garbage collection algorithm are given.

An interesting new option was introduced in this JVM : `-compactInterval=<num>`. It forces a Java heap compaction when the time since the previous garbage collections is less than `num`, where `num` is a time specified in milliseconds. Use of `compactInterval` minimizes heap fragmentation by compacting the Java heap before the GC algorithm would normally trigger compaction. Minimizing fragmentation extends the time period between required garbage collections, but incurs the cost of the more frequent compactions. Generally, the net effect on the program is reducing the total time required to do garbage collections by the application. The garbage collector does not have a specific thread. Instead, whichever thread triggers the collection is the one that is used to perform the collection. The asynchronous collector (available only on green-threaded JVMs with the `-asyncgc` flag) is basically a very low priority thread which periodically calls `java.lang.Runtime.gc()`.

ChaiVM

The other HP virtual machine, made for embedded systems is called ChaiVM [31]. It is fully compatible with Java Virtual Machine Specification and provides support for the Java Native Interface (JNI). Chai VM has a concurrent garbage collector that uses mark-sweep algorithm that is capable of running in the background thus minimizing overhead on memory operations. It ensures simultaneous operation without preempting application code. It is available for download under a license agreement. Recently, HP released MicrochaiVM, a virtual machine for mobile devices [32], with no details on possible garbage collector.

3.2.7 IBM Runtime Environment for Windows, Java Technology Edition, Version 1.1.7

IBM has developed its own version of JDK based on Sun's JDK. The latest version is 1.1.8 (Developer Kit and Runtime Environment). It includes the IBM just-in-time compiler version 3.5. It also includes a garbage collector, but no details on its algorithms are given. In the FAQ, however, it is explained that the Java heap organization has been optimized to reduce fragmentation, thus increasing memory utilization efficiency and reducing garbage collection activity. Also, the improvements have reduced the duration of the garbage collection-related application program delays sometimes referred to as pause-times; such delays are troubling for transactional systems and their reduction is an important advance.

Different versions of IBM JDK are freely available for download on its Web site [33]. The following platforms are covered : Linux, Windows, AIX, AS/400, OS/2, OS/390 and VM/ESA.

3.2.8 Japhar

Japhar is a Java virtual machine developed by Hungry Programmers [34]. It is a clean-room implementation, compatible with Java 1.1.5. It is licensed under the GNU LGPL, so it can be freely distributed. There are patches for Win32 and Solaris. No details are given on the garbage collection, except that it has been worked on. We have not been able to get the details in personal communications with Japhar developers. Our assumption is that, for the moment, Japhar does not implement a garbage collector.

3.2.9 JOVE

JOVE [35] is not a virtual machine in the sense that the term is most commonly used, but we mention it here for its garbage collection system. It does not interpret or translate Java bytecode as the Java program executes. Instead, JOVE is more like a standard compiler. Using JOVE, a Java program is translated to optimized native machine code before the program is distributed to its users. JOVE differs from compilers for languages such a C in that it does not process Java source code. Instead, JOVE accepts as input the same Java class files that would be used with a Java virtual machine. Because JOVE operates upon class files it preserves Java's "write once, run everywhere" characteristic. An application is created in class file format using any conventional Java IDE. Those class files may then be distributed to run on any platform using a Java virtual machine. The very same class files can be processed by JOVE to create an optimized native program for specific platforms.

Programs deployed using JOVE do not use a runtime virtual machine. Instead, such programs utilize the services of the JOVE runtime environment which consist of a small set of subroutines that define the runtime representation of objects and provide services for creating and managing objects. JOVE object references are represented by direct pointers to objects. A typical JOVE object requires only 4 bytes of overhead memory over and beyond the actual fields of the object that were defined by the Java programmer.

The JOVE runtime is structured around a high-performance, multi-generational garbage collector. It is a precise, copying, multi-generation collector [36]. The number of generations, size of each generation, and promotion thresholds are all dynamically adaptable to the behavior of individual programs. Special handling is provided to minimize copying overhead for large objects. Inter-generation references are tracked by remember sets. The JOVE garbage collection system is claimed to be significantly faster then existing collectors.

3.2.10 Java Virtual Machines for Embedded Systems

Besides already mentioned ChaiVM from HewlettPackard, there is a number of other Java virtual machines for embedded systems. Below we present four virtual machines from Sun (JavaCard, KVM, PersonalJava and EmbeddedJava), and JVMs developed by Charis (picoVM), Newmonics (PERC VM) and Oberon (JBed).

Sun's JavaCard

Java Card technology [37] enable using of Java technology on smart cards. The virtual machine (VM), the language definition, and the core packages have been made more compact to bring Java technology to the resource-constrained environment of smart cards. The Java Card specification does not mandate the garbage collection. Its white paper further mentions that more memory on the smart card will also enable more complete implementations of the Java Card specification, plus additional benefits outside the standard, such as automatic garbage collection and advanced services similar to Java's object-oriented Remote Message Invocation to allow Java Cards to communicate more easily with Java terminals by relieving programmers from dealing with low-level protocols.

Sun's KVM

K Virtual Machine is a new virtual machine from Sun, highly optimized for small-memory, limited-resource connected devices such cellular phones, pagers, PDAs, set-top boxes, and point-of-sale terminals.It is implemented in C programming language, and has a memory footprint of the virtual machine core in the range 40 kilobytes to 80 kilobytes (depending on the target platform and compilation options). It is part of Java 2 Micro Edition (J2ME) Connected Limited Device Configuration and is available for download from its site [38].

This virtual machine has a simple, handle-free, non-moving, single-space mark-sweep garbage collector. It operates with heap sizes of just a few tens of kilobytes.

Sun's EmbeddedJava and PersonalJava

The EmbeddedJava [39] and PersonalJava [40] are both designed to accommodate devices with severely limited memory. The difference is that EmbeddedJava does not include libraries for web-connection. Developers can use the EmbeddedJava application environment to create a variety of products including non web-based mobile phones, network routers and switches, industrial controllers, printers etc., whereas PersonalJava can be used for building web-connectable consumer devices for home, office, and mobile use. such as set-top boxes or web phones.

Both of these application environments include a Java virtual machine, but no details are given on its garbage collection algorithms. We suppose that they use the same system as Java1.1. (see Section 3.2.1). For now, there is no real-time support in these virtual machines, but Sun released a specification of a real-time system [41], so we suspect that it will be soon included in these virtual machines. It is interesting to cite the part of the specification that deals with memory management :

" ... This section contains classes that:

- Allow the definition of regions of memory outside of the traditional Java heap.

- Allow the definition of regions of scoped memory, that is, memory regions with a limited lifetime.
- Allow the definition of regions of memory containing objects whose lifetime matches that of the application.
- Allow the definition of regions of memory mapped to specific physical addresses.
- Allow the specification of maximum memory area consumption and maximum allocation rates for individual real-time threads.
- Allow the programmer to query information characterizing the behavior of the garbage collection algorithm, and to some limited ability, alter the behavior of that algorithm. ... "

The real-time garbage collection was not a subject of this work, but it is interesting to see that Java community had to let go of the heap-only allocation, as well to introduce C-like static memory in order to accommodate real-time systems.

An interesting link on Sun's Java Web site points to PersonalJava emulator [42]: a tool that verifies whether a Java application made in JDK would work in PersonalJava application environment.

The source code for both EmbeddedJava and PersonalJava is provided as part of the Sun Microsystems Community Source Licensing program.

PERC Virtual Machine 1.0

PERC [43] was designed specifically to meet the needs of embedded real-time developers. The PERC virtual machine is a clean-room implementation of the Java platform, independently developed, but fully compatible with the Java Virtual Machine Specification. The PERC virtual machine uses accurate incremental real-time defragmenting garbage collection. No details on the algorithm are given except that it uses a hybrid of classic garbage collection algorithms. The garbage collector is scheduled aggressively, so as to stay ahead of application requirements to allocate memory. The amount of CPU time dedicated to garbage collection is configured as a function of the system workload. There are no memory leaks resulting from conservative scanning and live objects are copied incrementally to contiguous memory locations so as to coalesce free segments. After garbage collection is preempted by higher-priority tasks, garbage collection resumes where it left off. PERC's API aggressively schedules garbage collection. A programmer can tell PERC the amount of memory needed and PERC treats garbage collection as a real-time task. This ensures that a temporary lack of memory does not stall forward progress of real-time tasks.

A free PERC 2.2 Evaluation Kit is available upon request. Academic licenses are available at a discounted price.

Charis pico Virtual Machine (pVM)

Charis [44] has developed a full function Java-compliant Virtual Machine (pico Virtual Machine) of less than 25K byte in size. It is used in micro-controller based embedded systems employed in pagers, smart card readers, cell-phones, hand-held toys, intelligent appliances and hand-held terminals.

The pico Virtual Machine supports garbage collection. It offers support for multi-segment memory management (buddy system) with simple incremental (usage-counter based) garbage collection. There is, also, a support for optional mark and sweep on demand garbage collection. pVM minimizes memory usage by sharing the structures between the memory allocator and garbage collector. This sharing allows for collection without using the stack when traversing object references.

A demo is available upon request.

Jbed

Jbed is a clean-room application environment for embedded systems. Jbed white paper [45] explains the differences between above-mentioned JVMs and Jbed, as well Jbed's place in the world of embedded systems :

> "In particular, JavaCard and Embedded Java appear as candidates for using Java in embedded and real-time systems. JavaCard is too limited for most applications, it doesn't support threads, garbage collection, floating point numbers, etc. Embedded Java on the other hand is still too close to the original Java platform and its slightly reduced Personal Java version. It doesn't address any real-time issues,. such as real-time memory management, real-time exception handling, or real-time thread management. JBed is the missing link between JavaCard and Embedded Java. It addresses the mentioned problems of Java by providing real-time memory management (garbage collection!), a library that allows to write device drivers completely in Java, a process peripherals framework, a hard real-time thread library, and of course the dynamic loading and replacement of code."

As for garbage collection Jbed runtime system supports real-time memory allocation and real-time garbage collection.

3.2.11 Conclusion

Different JVMs use very different (or none) garbage collection algorithms. It seems that Sun's HotSpot has the most sofisticated garbage collection system, although there are not many reports on its efficiency. Many JVMs use some kind of real-time algorithm : most of those JVMs are embedded, i.e. support applications made for a limited memory environment. Some JVM specifications mention garbage collection but give no details on its algorithms. Several JVMs implement conservative garbage collection (Kaffe and partly Sun's JVM), and some of them do not mandate any garbage collection (Japhar, JavaCard). Table 3.1 presents all found JVMs and their garbage collection algorithms.

JVM	Garbage Collection algorithm
Sun's JDK1.0 - JDK1.1 - JDK1.2.2 (classic)	Compacting, mark-sweep collector with some conservative scanning
Sun's Java 2 Production Release for Solaris	Generational copying collector Standard mark-compact collection algorithm
Sun's JDK1.2.2 (HotSpot)	Generational copying collector Standard mark-compact collection algorithm Fully incremental "train" algorithm
Kaffe	Non-incremental, non-generational, conservative mark-sweep collector with a Boehm-like allocator
LaTTe	Partially conservative mark-sweep garbage collector
Mach J	Real-time incremental garbage collector
DynaFlex (TowerJ 3.0)	Supports different gc schemes (algorithms not detailed)
Hewlett-Packard's HP-UX v1.1	Supports garbage collection (algorithm not detailed)
Hewlett-Packard's ChaiVM	Concurrent mark-sweep garbage collection
IBM Runtime Environment for Windows	Supports gc (algorithm not detailed)
Japhar	No garbage collection
JOVE	Copying, multi-generation collector
Sun's JavaCard	No garbage collection
Sun's KVM	Simple, handle-free, non-moving, single-space mark-sweep garbage collector
Sun's EmbeddedJava and PersonalJava	v. Sun's JDK1.0
PERC VM	Accurate incremental real-time defragmenting garbage collection
Charis pico Virtual Machine (pVM)	Multi-segment memory management (buddy system) with simple incremental (reference-counting) garbage collection Support for optional mark-sweep on demand garbage collection
Jbed	Real-time garbage collection

Table 3.1: JVMs and their corresponding garbage collection algorithms

3.2.12 Which of Them to Test

Is it available ?

Not all of mentioned Java virtual machines are interesting or available for testing. First of all, we excluded for the moment JVMs for embedded systems, as we wanted to have no memory limitations. Also, there are some Java virtual machines that are not available on the Internet (either the licencing fee was very big, or there was no response to the mail asking for an evaluation copy). The list shortened also because of the problems with installation (see below).

Where does it work ?

Sun's JDK is naturally the most portable and certainly available, but there is also a limit here : Hotspot is available only for Win 32 and Solaris. So testing on Linux was out of question. IBM's JDK 1.1.7 works only on Windows, whereas LaTTe works only on Solaris. Kaffe and Japhar (Japhar is interesting because it is open source, which means that the garbage collector could be easily added to it) do not compile on Win32. So that leaves Solaris. Japhar still doesn't compile on Solaris. So it leaves us with JDK (1.1, 1.2 (classic and hotspot)), Kaffe and LaTTe as Java virtual machines available for testing.

3.3 Choice of benchmarking application

3.3.1 What Kind of Application to Look for ?

An application that is to be used to test a garbage collector should allocate a lot of objects, and preferably have a well-documented allocation behaviour. The application should execute on all of chosen virtual machines.

3.3.2 What Sorts of Applications Are Available ?

There are two kinds of interesting applications. The first one is an application that is at the same time a profiler. These are usually applets, and not applications, which call several programs subsequently and measure the time of execution of each of them, along with some other parameter (quantity of free memory, for instance).

The other type of application is just some application that allocates fairly enough to be used for testing garbage collectors. We mention those that we found interesting, although some of them were not used in our benchmarks for portability reasons.

The following sections present mentioned applications.

Applications-Profilers

Here we mention some applications/applets that are at the same time profilers. These are : CaffeineMark, the Benchmark Applet, UCSD Benchmarks for Java and Java Grande Forum Benchmark Suite.

CaffeineMark 3.0 CaffeineMark [46] is an applet that runs a series of tests, which measure the speed of Java programs running in various hardware and software configurations. CaffeineMark scores roughly correlate with the number of Java instructions executed per second, and do not depend significantly on the the amount of memory in the system or on the speed of a computers disk drives or Internet connection. CaffeineMark uses nine tests to measure various aspects of Java virtual machine performance. Each test runs for approximately the same length of time. The score for each test is proportional to the number of times the test was executed divided by the time taken to execute the test. This test measure the overall performance of a JVM, but not the performance of the garbage collector itself.

The Benchmark Applet The Benchmark Applet [47] measures the time it takes to do an operation thousands (or even millions) of times, subtracts the time spent doing operations other than the test, such as the loop overhead, and then uses this information to compute how long each operation took. It runs each test for approximately one second. In an attempt to eliminate random delays from other operations the computer may perform during a test, it runs each test three times and uses the best result. There are 10 individual benchmarks included in The Benchmark Applet, and it is up to the user to make a choice. Similarly as the previous one, this benchmark application does not have a possibility to test specifically the garbage collector.

UCSD Benchmarks for Java This is a collection of methods that test different aspects of JVM : garbage collection, method invocation, loops, etc. It can be downloaded and compiled, or it can be run as applet from its site [48]. On the contrary to the previous applications, UCSD Benchmarks include a garbage collection benchmark. Two types of GC measurements have been performed. The first requests that the runtime perform a full GC. Both the time and the space recovered (and perhaps added) are reported. The second "randomly" allocates several thousand objects into a small array (the time for looping and array access are probably a small part of the overall time.), causing most to be made available for reclamation by the garbage collector. Because, for example, Sun's Java runtime has an incremental collector, many of these may be collected without a full GC occuring. The authors state also that the garbage collection benchmark seems to have widely varying behavior depending upon the environment, including crashing (Sun's appletviewer) and thrashing/hanging (Netscape).

Java Grande Forum Benchmark Suite Java Grande Forum (JGF) Benchmark Suite [49] is a collection of benchmarking applications from EPCC, University of Edinburgh. The applications are grouped into three sections : low level operations, kernels and applications. Low level operation section, in version 1.0, contains *Garbage* benchmark, that assess the performance of the system garbage collector. Objects are created with a randomly chosen size in the range zero to (total available memory)/1000. Initially, sufficient objects are created to consume all available memory: this part is not timed. Subsequent object creation proceeds for a fixed time period. All objects are assigned to the same reference, so that all objects except the most recently created are available for collection. The number and total size of the objects collected is recorded. Performance units are references per second and bytes per second. The *Garbage* benchmark is excluded in the recent version of JGF Benchmark Suite.

Other benchmark application There is an excellent source of benchmark applications on the Java Grande Forum site [49], but hardly any of them is useful for garbage collection benchmarking.

Applications that allocate a lot of memory

Other possibility to test garbage collectors in Java virtual machines is to execute applications that allocate a lot of objects, and to use either a `-verbosegc` flag or some other method (see Section 3.4) to measure the performance of a garbage collector. We searched the Internet again to find the applications that meet our needs, and we found some on the benchmark sites. Here we mention CUP and JLex.

CUP CUP (Constructor of Useful Parsers) is a system for generating LALR parsers from simple specifications. It serves the same role as the widely used program YACC and in fact offers most of the features of YACC. However, CUP is written in Java, uses specifications including embedded Java code, and produces parsers which are implemented in Java. There are some grammars downloadable from the CUP site [50], which we used as an input.

JLex JLex [51] is a lexical analyzer generator, written for Java, in Java. The JLex utility is based upon the Lex (lexical analyzer generator) model. JLex takes a specification file similar to that accepted by Lex, then creates a Java source file for the corresponding lexical analyzer. We were not able to find a sample of the grammar that is large enough to cause garbage collection while executing JLex, so we mention it just as a reference.

3.4 Choice of profiling methods

After we decided on the applications to use, we had to choose the profiling method, i.e., the way to measure the performance of the garbage collector. We mention Java flags, Java methods, heap simulators and profilers.

3.4.1 Java Flags

One could say that using of Java flags aimed for profiling (Java application launcher `-prof` option) is the most universal choice, but unfortunately it is not. JDK changed its runtime options or the way the options work with every new version. Kaffe does not support `-prof` option. So, we usually used `-verbosegc` option, which is not implemented in the same way for all the JVMs, but still gives rather similar informations. For all the Java virtual machines, `-verbosegc` gives the amount of freed memory (in

KB/MB or %), and the total heap before and after each collection. All of them, except Kaffe, give the time needed for each collection. We instrumented Kaffe (with the help of LaTTe's code) to obtain timing information.

We mention a useful application for analysing the `-prof` option's output : Hyperprof [52].

Hyperprof This is a program written in Java that allows the user to view and analyse the execution profile of Java program. It parses java.prof files obtained using the `-prof` option to give a list of methods, list of methods that called selected method, list of methods that are called by selected method and memory used by program. The problem is that it works with JDK1.0 and JDK1.1, but not with Kaffe (which does not implement `-prof` option) or JDK1.2 (which changed the format of the .prof file).

3.4.2 Java Classes

Method `freeMemory()` of the class java.lang.Runtime returns the amount of free memory in the system. We could use it, along with the call to garbage collector (`System.gc()`) to measure the quantity of memory before and after the call to the application, or before and after the call to the gc (Fig. 3.4.2). The problem is that Kaffe does not implement this method the same way JDK does. Kaffe's method returns RAM (therefore before and after are same and constant always) and JDK's method returns free portion of the heap.

We can use a simple application that allocates an array of a given number of Integers, then nulls it and calls the garbage collector. Of course, this call to the garbage collector may not be the only one. The garbage collector can be called by the JVM itself, if the array is too big. This is not how the usual Java program works, there are surely some objects that die faster and other that die slower, but it can show us how the collector works in these cases. An example of such an application is shown in Fig. 3.4.2 below.

3.4.3 Some Kind of Heap Simulator

Some researchers use their own instrumentation to simulate heap allocation. Darko Stefanovic [53] uses an accurate simulator that models all heap objects and pointers among them. Jonathan Moore et al. [54] use Oscar, a GC testbed that compares GC performance across different languages. Both of them instrument a language implementation to gather snapshots of its heap. An instrumentation of a JVM requires the source code of the JVM, which was not available for all tested JVMs, so we dropped this approach.

3.4.4 Profilers

There is a number of Java profilers available. They are usually expensive, but there are almost always free, time-limited evaluation copies. The most popular are : OptimizeIt [55] and JProbe [56]. All of them have a user-friendly graphic interface, with plenty of

```
import java.io.*;
classMemoryAndTime {
     publicstatic void main(String[] args) {
        int count;
        Runtime rtObj = Runtime.getRuntime();
     // — begin time
        long begTime = System.currentTimeMillis();
        long fm1,fm2, endTime;
        if ( args.length < 1 ) {
                 count = 1000;
        } else {
                 count = Integer.parseInt(args[0]);
        }
        System.out.println("Memory printed in K bytes ...");
        System.out.println("Integers to be allocated:  " + count);
     // — create an array
        Integer someints[] = new Integer[count];
     // — total heap memory
        System.out.println("Total memory        :  " +
                                                 rtObj.totalMemory()/1024);
     // — free memory after the array creation
        System.out.println("Free memory   :  " + rtObj.freeMemory()/1024);
     // — call to garbage collector
        rtObj.gc();
     // — free memory after garbage collection
        System.out.println("Free memoryafter GC   : " +
                                                rtObj.freeMemory()/1024);
     // — allocate array members
        for (int i = 0; i ¡ count; i++ )
                          someints[i] = new Integer(i);
     // — free memory after initialization of array members
        System.out.println("Free after alloc of ints:  " +
                                                rtObj.freeMemory()/1024);
     // — nulls the array
        for (int i = 0; i < count; i++ ) someints[i] = null;
     // — call to garbage collector
        rtObj.gc();
     // — end time
        endTime = System.currentTimeMillis();
     // — free memory after the second garbage collection
        System.out.println("Free memoryafter GC : " +
                                                rtObj.freeMemory()/1024);
     // — total time
        System.out.println("Total time :" + (endTime - begTime) +  "ms");
     }
}
```

Figure 3.1: Simple Java code for garbage collector testing

functions that are very useful. Unfortunately, they are always destined to a particular JVM. JProbe is compatible with Java2 (JDK1.2), because JProbe use its profiling API, which is not present in JDK1.1. As for JDK1.1, it uses a specially instrumented version of JDK1.1, that is included in JProbe. Normal JVMs 1.1 are not compatible with JProbe, because of the lack of that instrumentation. Logically, it does not work with Kaffe nor with LaTTe.

OptimizeIt for Solaris is compatible with Sun's JDK1.2 reference implementation, but not with Sun's JDK1.2.2, nor with Kaffe or LaTTe (OptimizeIt for Win32 does not work with IBM JDK1.1.7 either).

For these evident reasons, we did not make use of the profilers.

3.5 Our Choice

We chose to test JDK1.2.2-0.0.1 (classic and Hotspot), JDK-1.2.2-05, JDK1.1.3, Kaffe and LaTTe on Solaris using a Java flag **-verbosegc** while executing either CUP separately, or CUP installed in a piece of code that verifies the quantity of free memory.

This choice is obvious from the above analysis : Kaffe compiles on Solaris, and not Win32; Hotspot is available for Solaris and Win32; LaTTe is made exclusively for Solaris; the rest of JVMs either do not work on Solaris (IBM's JDK1.1.7) or are not available for testing. Neither of available profilers recognizes Kaffe nor LaTTe. Most benchmarking applets/application do not include garbage collector tests.

CUP was compiled using JDK1.2.1, and the class files obtained that way are used for testing all JVMs.

All the test are done on Sparc SunOS 5.6.

3.6 Benchmarks

3.6.1 verbosegc Output

Here we present the examples of the **verbosegc** outputs for all the Java virtual machines tested. It is interesting to compare the information obtained using this flag for each JVM.

JDK1.2.2-05

JDK1.2.2-05, which is also known as ExactVM or ResearchVM, has two **-verbosegc** options. One is a standard option, which gives less verbose output, stating the gc time, the total heap, and the percentage of the free portion of the heap :

```
GC[0] in 23 ms: (8Mb, 75% free) -> (8Mb, 94% free)
```

The other one is a triple **-verbosegc** option, i.e., "**java -verbosegc -verbosegc -verbosegc ...**". It gives more verbose output, and the details on the collections of each of its two generations. For the Gen 0, which has two semi-spaces and uses a copying collector, it gives the size of the semi-spaces, its occupancy and the free memory. Fig. 3.2 presents the same output as above, but with the triple **verbosegc** option.

```
Starting GC at Wed Apr 19 18:16:38 2000; suspending threads.
Gen[0](semi-spaces): size=4096kb(50% overhead), free=0kb, maxAlloc=0kb.
  space[0]: size=2048kb, free=0kb, maxAlloc=0kb.
  space[1]: size=2048kb(100% overhead), free=0kb, maxAlloc=0kb.
Gen0(semi-spaces)-GC #1 tenure-thresh=31 26ms 0%->77% free
Gen[0](semi-spaces): size=4096kb(50% overhead), free=1586kb, maxAlloc=1586kb.
  space[0]: size=2048kb(100% overhead), free=0kb, maxAlloc=0kb.
  space[1]: size=2048kb, free=1586kb, maxAlloc=1586kb.
resuming threads.
GC[0] in 28 ms: (8Mb, 75% free) -> (8Mb, 94% free)
Total GC time: 28 ms
Processing 26 reference objects.
++ GC added 26 finalizers++ Pending finalizers = 26
++ Finalizer Q = 0
```

Figure 3.2: **-verbosegc** output for JDK1.2.2-05 JVM

JDK1.2.2-001 classic

JDK1.2.2 classic has the same output, and consequently and probably the same garbage collector system as the previous versions of JDK (v. JDK1.1.3). It gives a number of freed objects, freed memory in bytes, the time of each phase of garbage collection, and the percentage of the free memory (see Fig. 3.3).

```
<GC: managing allocation failure: need 1032 bytes, type=1, action=1>
<GC: 0 milliseconds since last GC>
<GC: freed 12345 objects, 530552 bytes in 13 ms, 63% free (531272/838856)>
  <GC: init&scan: 0 ms, scan handles: 8 ms, sweep: 5 ms, compact: 0 ms>
  <GC: 0 register-marked objects, 13 stack-marked objects>
  <GC: 0 register-marked handles, 68 stack-marked handles>
  <GC: refs: soft 0 (age >= 32), weak 0, final 17, phantom 0>
```

Figure 3.3: **-verbosegc** output for JDK1.2.2-001 classic JVM

JDK1.2.2-001 hotspot

Hotspot has a rather non-verbose **-verbosegc** option. Nevertheless, it gives enough information : allocated memory before and after the collection, total heap, and the gc time :

```
[GC 2048K->422K(5184K), 0.0456488 secs]
```

As Hotspot uses a generational gc, when the whole heap is collected it is mentioned:

```
[Full GC 5567K->3564K(8640K), 0.2738012 secs]
```

When the incremental collections is enabled the output has the same form.

JDK1.1.3

JDK1.1.3 gives the output similar to the one given by JDK1.2.2 classic (Fig. 3.4).

```
<GC: managing allocation failure. need 24 bytes, type=2, action=1>
<GC: freed 17597 objects, 378152 bytes in 18 ms, 66% free (560112/838856)>
  <GC: init&scan: 0 ms, scan handles: 11 ms, sweep: 7 ms, compact: 0 ms>
```

Figure 3.4: -verbosegc output for JDK1.1.3 JVM

Kaffe

Kaffe's verbosegc output gives the total heap, memory allocated before and after the collection, the percentage of the heap that is free, but it lacks the gc time (Fig. 3.5).

```
<GC: heap 5120K, total before 4705K, after 2142K (104084/33383 objs)
 58.2% free, alloced 17130K (#131174), marked 750K, swept 2563K (#70701)
 42 objs (1K) awaiting finalization>
```

Figure 3.5: -verbosegc output for Kaffe JVM

In order to have the timing information, we instrumented Kaffe, by incorporating the code from LaTTe that gives the gc time. The changed output is showed in Fig. 3.6

```
<GC: walk root = 0 ms>
<GC: walk mark stack = 188 ms>
<GC: sweep = 70 ms>
<GC: total = 260 ms>
<GC: heap 5120K, total before 4694K, after 2174K (102770/33545 objs)
 57.5% free, alloced 17344K (#133159), marked 726K, swept 2519K (#69225)
 46 objs (1K) awaiting finalization>
```

Figure 3.6: More verbose -verbosegc output for Kaffe JVM

LaTTe

LaTTe has rather verbose -verbosegc output. It shows time for each garbage collector's phase, the total heap and the amount of freed memory (Fig. 3.7).

```
<GC: walk root = 1 ms>
<GC: walk mark stack = 1 ms>
<GC: process finalizers = 0 ms>
<GC: sweep = 3 ms>
<GC: total = 7 ms>
<GC 1: heap 8192K, total 8192K, marked 171K, freed 5323K, fixed 2535K>
```

Figure 3.7: -verbosegc output for LaTTe JVM

Analysis and presentation of the verbosegc output

All the verbosegc outputs from different JVM give the following numbers : amount of total heap, amount of freed memory and total gc time. When executing a benchmark application, we use the verbosegc output to show : the total gc time for each JVM with standard options, gc time and number of collections as a function of the initial heap size (set by the -ms option), and the total gc time as a function of maximal heap size found by previous analysis.

3.6.2 CUP Benchmarks

The first application that we used was CUP. CUP uses as input a Java grammar; we used java10.cup, a file that contains the Java 1.0 grammar. Here we present the results obtained. First we executed the application and measured the total gc time for each JVM (see Fig. 3.8).

We can see that, with default start heap size, specific for each JVM, JDK1.2.2 (ExactVM) has the best performance, and Kaffe is the least performant. LaTTe's garbage collector performs surprisingly well.

In order to compare the obtained results more objectively, we ran the application changing the initial heap size (using the -ms option of java runtime). We were able to find the heap size for each JVM for which only 1 or 0 garbage collections are performed, and the gc time is, therefore, minimal. Fig. 3.9 shows gc time as a function of the initial heap size for each JVM.

Minimal values for gc time correspond almost always to the minimal heap for which only one garbage collection is performed. The only exception is JDK1.2.2 (ExactVM) where the number of garbage collections is always 12, independently on the heap size. The reason is its generational garbage collection algorithm, which forces frequent collection in the nursery, whose size is constant (i.e. independant on the total heap size). In order to verify the results for gc time obtained by standard execution (with the

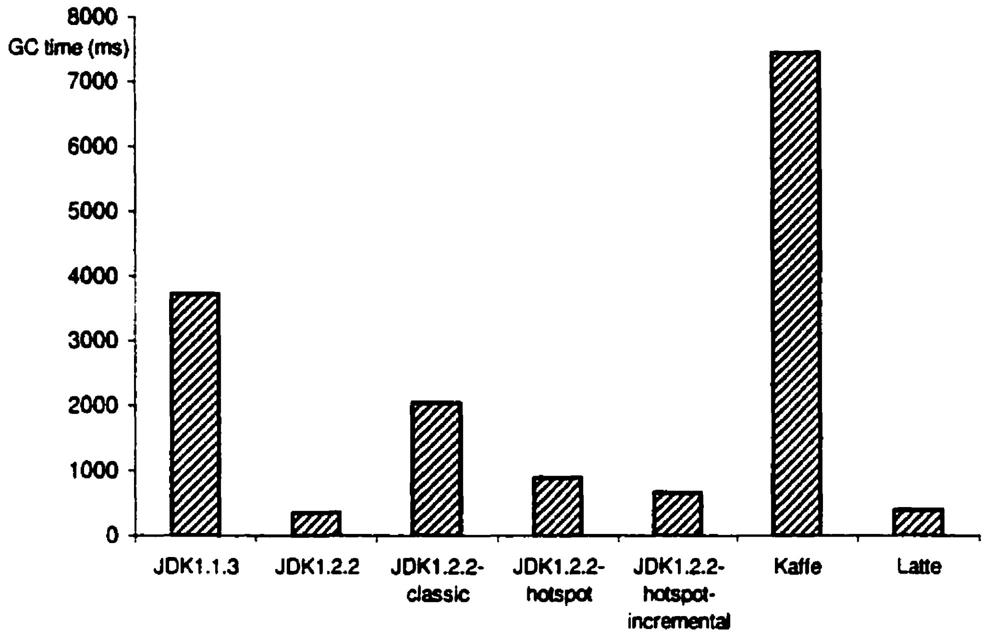

Figure 3.8: GC time with the default heap size for each JVM

initial heap size being a default value, different for each JVM) (see Fig. 3.8), we compared the minimal values for gc time for each JVM, and its corresponing heap size. The result is shown in Fig. 3.10.

If we compare graphics in Fig. 3.8 and Fig. 3.10, we can see that, when we exclude the heap size factor, LaTTe performs the best, and Hotspot, JDK1.2.2 classic, JDK1.2.2 (ExactVM), and JDK1.1.3 follow it closely. Kaffe still has the worst performance, but the difference is somewhat reduced.

3.6.3 MemoryAndTime Benchmarks

In section 3.4.2 we mentioned that we could use a simple application that allocates an array of Integers, to measure the garbage collector activity. We used the class MemoryAndTime (see Fig. 3.4.2) creating an array of 1000000 Integers. While allocating the array, each JVM made several calls to the garbage collector in order to find the needed memory; the heap is therefore augmented several times. At the end, when the array is nulled out, we placed a call to the garbage collector to see what time it takes to free the allocated memory (we used the `-verbosegc` output). Fig. 3.11 shows the obtained results : number of KB deallocated per ms of GC time.

This kind of benchmarking is not at all the way real Java programs deallocate memory, and that should be kept in mind. Under these conditions, Hotspot shows the best performance, with LaTTe close by, incremental Hotspot and the ExactVM are rather efficient also. JDK1.1.3, JDK1.2.2-classic and Kaffe have a lot worse performance.

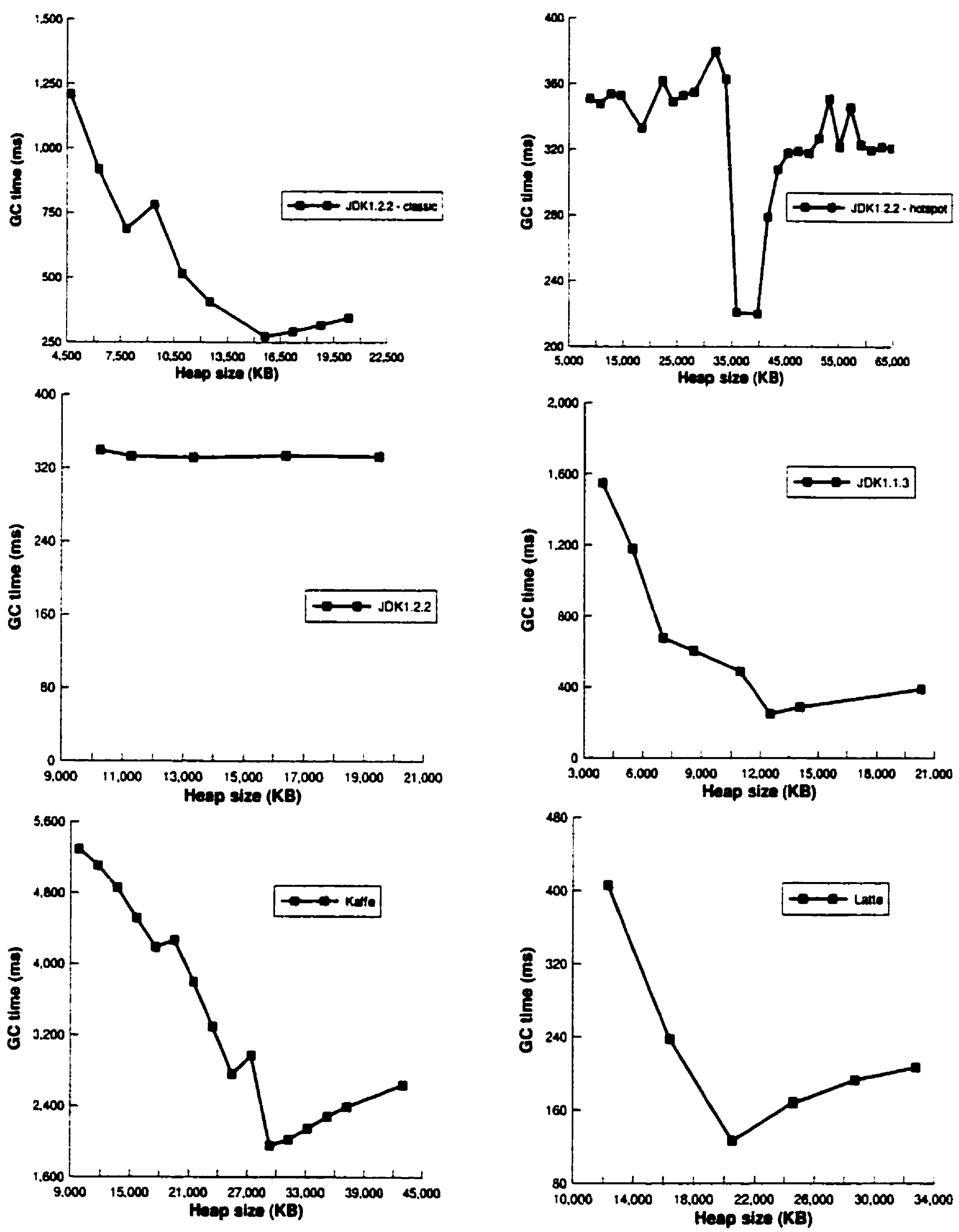

Figure 3.9: GC time as a function of the initial heap size

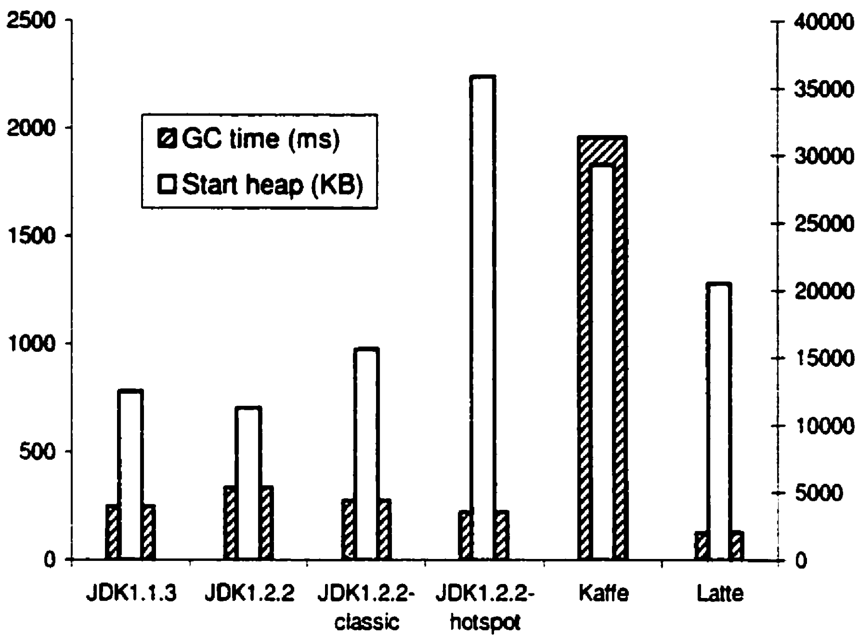

Figure 3.10: Minimal GC time and the corresponding initial heap size

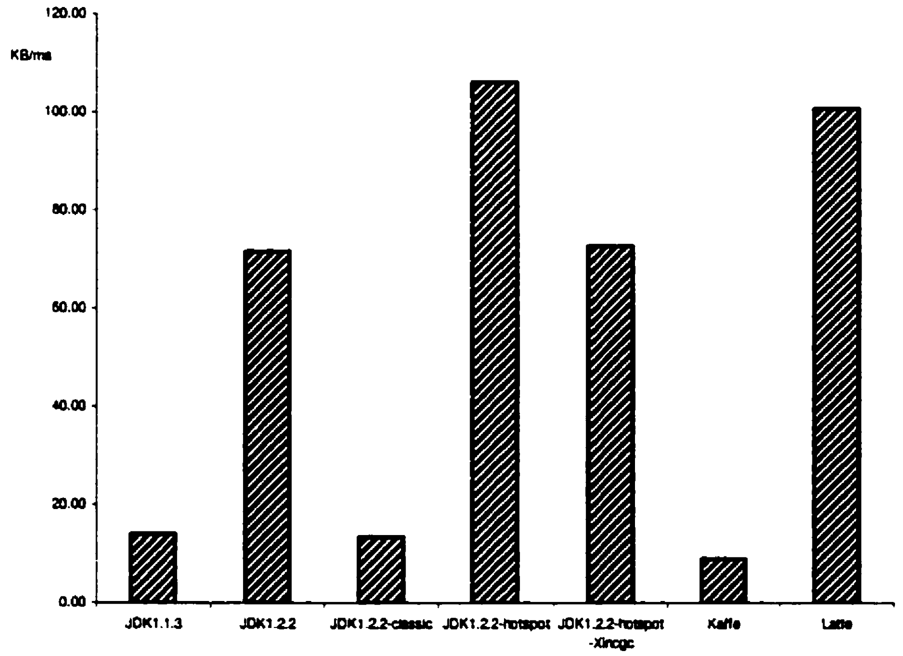

Figure 3.11: Performance of the garbage collector for each JVM for MemoryAndTime application

3.7 Conclusion

The choice of Java virtual machines, applications, profilers and operating systems to test garbage collectors is not always obvious. It depends on the virtual machine's availability for a specific operating system(s), or the application's or profiler's availability for a specific JVM. We have chosen to test different Sun's JVMs, as well as Kaffe and LaTTe on Solaris, with `-verbosegc` output as a source for benchmarking results.The results showed that LaTTe, Sun's Hotspot and Sun's ExactVM have the best performing garbage collector systems.

As LaTTe's garbage collector performed very well, and as its source code is available, we analyzed its code to better understand it. This was important both for understanding the garbage collector's algorithm and for understanding the memory management system as a whole, thus making it easier to introduce different algorithm, and compare it to the present one. In the next chapter, we analyze the source code of LaTTe's memory management system.

Chapter 4

The LaTTe Java Virtual Machine

4.1 Introduction

LaTTe garbage collection system showed very good performance, so we wanted to go into details of its algorithm, to be able to explain that performance. First, we give a short overview of LaTTe, based on the information from its web site [2], and then we present LaTTe's memory management.

4.2 Overview

LaTTe was created by the MASS (Micro-Architecture and System Software) Laboratory of the School of Electrical Engineering at Seoul National University [57], as joint work with the VLIW research group at IBM T.J. Watson Research Center [58].

It was built starting with the code from Kaffe 0.9.2, a freely available Java virtual machine [27]. However, the core parts were re-written : the bytecode execution engines, the garbage collector, the exception manager, and the thread synchronization mechanism. The execution engines of LaTTe (i.e., the JIT compiler and the interpreter) were written from scratch. Both compiler and interpreter are more elaborate than Kaffe's, since LaTTe does not provide multi-platform support, which makes it easier to implement more powerful execution engines. Additionally, the JIT compiler and the interpreter can be used concurrently in LaTTe, while Kaffe can use only one of the two execution engines. As for the garbage collector, LaTTe uses a similar algorithm as is used by Kaffe : a mark-sweep garbage collector. LaTTe's collector has, however, completely different internal structure, and the implementation used in LaTTe is a great deal faster (see Section 3.6). LaTTe uses Kaffe's user-level thread system with little modification, but the thread synchronization mechanism supporting Java monitors is newly designed to be much faster than that of Kaffe. This new design of the synchronization primitives results in a different object model.

However, compared to other commercial Java virtual machines, LaTTe has the following lack of features:

- No AWT or Swing.

- Not Java 2 (only supports 1.1).
- No bytecode verifier.
- Lacks JNI support.
- Incomplete class library.
- No support for JAR or compressed ZIP archives.

4.3 Memory Management in LaTTe

LaTTe's memory management changed a little from the version that we tested. The new version, 0.9.1 uses an improved garbage collection system. The details about LaTTe's memory management are given in [59]. Here we present a shorter description based on Chung's paper [59], as well as our ideas and further explanations, along with a thorough code analysis. Our idea was to introduce a new garbage collector in LaTTe, or to instrument the existing one, so the code analysis was a necessary step.

The following two sections describe, respectively, memory allocation and garbage collection in LaTTe. In order to make the code fragments easier to read, we present in Table 4.1 the most used types and macros.

4.3.1 Memory Allocation

LaTTe manages three types of heaps separately: a small object area, a large object area, and an explicitly managed heap. The small object area contains objects that are smaller than one kilobyte, while the large object area contains objects that are larger than one kilobyte. Objects in both of these heaps are deallocated automatically by the garbage collector when no longer needed. The explicitly managed heap, on the other hand, contains objects that must be deallocated manually by the virtual machine programmer. Such explicitly managed objects include class objects, register map tables, code fragments, etc. Allocating explicitly managed objects, or "fixed" objects, never triggers a garbage collection. Memory for these heaps is obtained from the operating system by the region manager.

Region manager

Each heap allocates memory through the region manager, by allocating regions (`gc_region`) of about 2MB. Each region has its type for distinguishing different heaps. Region information is held in the region table (`struct gc_region *regions[]`), which holds the information sorted in order of increasing addresses. The size of the region table array is 1024, which means that the total available memory would be about 2GB, which seems more than enough. The table is searched using binary search to find a region that contains a particular address.

Type/Macro	Definition		Source file
SIZE_MASK	~ 0x7		gc.c
HEADER(p)	(gc_head*)(p) - 1		gc.c
	struct gc_stats		
	size_t small_size, large_size, fixed_size	Sizes of various heap areas	
	size_t small_alloc, large_alloc, fixed_alloc	Various allocation statistics	
gc_stats	size_t small_marked, small_freed, large_marked, large_freed	Garbage collector statistics	gc.c
	int iterations	Number of GC done so far	
	double mark, sort, sweep, gc, total_mark, total_sort, total_sweep, total_gc	Timings for the garbage collector	
SIZE(p)	*HEADER(p) & SIZE_MASK		gc.c
MEMALIGN	sizeof(double)		gc.h
MAX_SMALL_OBJECT_SIZE	1024		gc.h
ROUNDUPALIGN(V)	(((uintp)(V) + MEMALIGN - 1) & -MEMALIGN)		gc.h
GC_LOCK()	intsDisable()		gc.c
GC_UNLOCK()	intsRestore()		gc.c

Table 4.1: The most used types and macros

Variable/Function	Description		Source file
`GC_REGION_TABLE_SIZE`	1024		gc.c
`gc_region`	struct gc_region `void *start, *end`	address range of region	gc.c
	`int type`	type of memory contained in region (type of heap)	
	`struct gc_region *prev, *next`	links for a doubly-linked list of regions	
	`void *data`	other data used by external components	
`static struct gc_region *regions [GC_REGION_TABLE_SIZE]`	region table		gc.c
`static struct gc_region* region_allocate(size_t, int)`	allocates a new region		gc.c

Table 4.2: Important variables and functions for region manager

As it is a mark-sweep collector, there is little chance that a region would be completely free, so freeing of regions is not yet implemented. The same goes for region merging.

Table 4.2 shows data structures and functions used for region manager.

Small object area

Objects that are automatically managed and smaller than one kilobyte are allocated in the small object area. They are allocated using lazy worst fit. In fact, objects are allocated using pointer increments (using `small_cursor` and `small_bound` pointers in a function `void* gc_malloc_small (size_t size)`) and if the allocation pointer would go over the bound pointer, the worst fit is used to find a new free space area (function `void* slow_small_allocate (size_t)`). Worst fit is accommodated by taking the first free memory chunk (`gc_small_chunk`) in a free list (`small_chunks`) sorted in decreasing order of size. The free list is built after the sweep phase. Worst fit has its advantages and disadvantages : a single comparison suffices to find out whether there is a node in the free list that satisfies the memory demand (first fit and best fit require many comparisons); on the other hand, worst fit results in more fragmentation, which can lead to smaller heap sizes. If there is no free chunk available, the function `void* get_small_block(void)` is called, and the allocation is made from the newly allocated block from the list of blocks (`small_blocks`). And if the list of blocks is empty, the small heap is expanded (new region allocated and put in the region list) by calling `void expand_small_area(void)`.

Important variables and functions for small object allocation are given in Table 4.3.

Variable/Function	Description	Source file
`static gc_head *small_cursor`	pointer to the free area	gc.c
`static gc_head *small_bound`	pointer to the end of the free area	gc.c
`gc_small_chunk`	struct gc_small_chunk `size_t size` size of chunk; `struct gc_small_chunk *next` pointer to the next chunk	gc.c
`gc_small_block`	struct gc_small_block `size_t size` size of memory occupied by adjacent blocks; `struct gc_small_block *next` pointer to the next group of blocks	gc.c
`void* gc_malloc_small (size_t size)`	allocate small object using pointer increments	gc.c
`static void* slow_small_allocate (size_t)`	allocator used when pointer increments fail to satisfy the request	gc.c
`static void expand_small_area (void)`	expand the small object area	gc.c

Table 4.3: Important variables and functions for small object allocation

Variable/Function	Description	Source file
`static struct gc_heap large_heap`	information for the large object area	gc.c
`large_set_node`	struct large_set_node `void *object` — the address of the object `struct large_set_node *prev, *next` — chained links	gc.c
`static int large_set_hash (void*)`	gets hash value for object	gc.c
`static void large_set_add (void*, size_t)`	adds a pointer to the object to the hash table	gc.c

Table 4.4: Important variables and functions for large object allocation

Large object area

Automatically managed objects larger than one kilobyte are allocated in the large object area (`large_heap`). Large object area uses the same allocator as the explicitly managed area (see function `void* mem_allocate (struct gc_heap*, size_t)` in Section 4.3.1). Upon allocation a pointer to the object is added to a hash table. All objects in this area are referenced by nodes a hash table (`large_set_node`) in order to support conservative pointer marking. The overhead associated with this is not large since all objects are larger than one kilobyte.

Important variables and functions for large object allocation are given in Table 4.4.

Explicitly managed heap

Manual memory manager is used both for explicitly managed heap and large object area. Data structures and functions used for manual memory manager can be found in Table 4.5. Heaps are distinguished by their types (field `type` in the structure `gc_heap`). The allocator uses segregated free lists (see Section 2.2.2). Memory is divided into free lists (field `lists`), one for each `size_class`, determined from the object size. For size classes the linear distribution is used for smaller sizes (up to 1024 b) and a power of two distribution is used for bigger sizes. Each free list is sorted in increasing order of size, which is ensured when nodes are entered into each free list (using the function `void insert_freelump(struct gc_heap, void*, int)`).

Whenever an object is to be allocated, first its size class is determined (function `int size_class (size_t)`). Then, the free list of that size class is searched in order to find a free lump that would fit the best (function `void* find_freelump (struct gc_heap, size_t int))`). If some space in the free lump is left free after the allocation, it is put back to the appropriate free list. If no free lump is found in that list, the search is continued in the next size class. If even after that the memory demand is not satisfied, the heap is expanded (function `int mem_expand (struct gc_heap*, size_t)`).

Variable/Function	Description	Source file
gc_heap	struct gc_heap `void *lists[SIZE_CLASSES+1]` — segregated free lists `size_t *size, *alloc` — placeholders for statistics `int type` — region type	gc.c
`static int size_class (size_t)`	calculate size class	gc.c
`static void* mem_allocate (struct gc_heap*, size_t)`	allocate memory from heap	gc.c
`static void insert_freelump (struct gc_heap*, void*, int)`	insert free lump into a free list	gc.c
`static void* find_freelump (struct gc_heap*, size_t, int)`	find a suitable free lump	gc.c

Table 4.5: Important variables and functions for manual memory management

4.3.2 Garbage Collection

LaTTe uses a non-incremental partially conservative mark and sweep garbage collector. Partially conservative means that LaTTe is not able to ascertain the types of local variables or stack operands. Thus, the garbage collector must be conservative with respect to the execution stack. In the case of heap objects, all objects have a class pointer which indicate their type. Thus the garbage collector can treat the heap in a type-accurate manner. A garbage collector that is partly conservative and partly precise in this manner is called a partially conservative garbage collector.

Garbage collection is done in a thread separate from the other normal threads (with for mutex `quickLock gcMan`). Table 4.6 lists the important garbage collector functions. The main garbage collector function is `gc_main()`, which calls the marking function `mark_phase()` and sweeping function `sweep_phase()`.

Marking phase

The functions used for the marking phase are listed in Table 4.7. Marking is done by first marking the objects referenced by the root objects, which are class objects and the execution stacks for each thread (using the function `void walk_roots ()`). Root objects are registered separately using `gc_attach()` and maintained using a linked list. To avoid overhead for linking each root object (which never reverts to normal, and therefore there is no bound to its number), root objects are grouped into root bundles

Function	Description	Source file
`void gc_invoke (int lack)`	wake up the garbage collection thread	gc.c
`void gc_main (void)`	the loop for the garbage collection thread	gc.c
`void mark_phase (void)`	execute the mark phase	gc.c
`void sweep_phase (void)`	sweep the heap for garbage	gc.c

Table 4.6: Important garbage collector functions

(`struct gc_root_bundle`) and a linked list of root bundles holds all root objects.

The objects thus marked are pushed onto the marking stack `void **mark_stack`. The size of the marking stack is checked and if it has less objects than a certain limit, a flag to selective sweeping is set. Then, the objects on the marking stack are "walked", that is, all the objects pointed-to by each object in the stack are marked and pushed onto the stack (it is a modification of Cheney's algorithm). If the number of marked objects is still less than a certain limit, the sweeping phase is entered. If not, the objects are marked by depth-first traversal using the marking stack : as object is popped from the stack, its children are put onto the stack, and the process repeated until the stack is empty. Marking function is made for each object of a certain class with `void makeWalkFunc (Hjava_lang_Class *class)`. It calls `void mark()` for each reference field (i.e. for each child of an object), so all the children are pushed onto the marking stack. Detection of stack overflow is done by an explicit bound check, before pushing an object on the stack. If the stack overflow occurs, it is handled by traversing all the objects in the heap, and marking the unmarked ones (functions `void walk_small_objects()` and `void walk_large_objects()`). This process is rather slow, but it seems to occur rarely, so it has no big impact on the performance.

Finding pointers

Before marking, the garbage collector makes sure that the pointer to the object to be marked points really to a heap object using the function `int is_object (void *p)`. This function first finds a region to which the object belongs. Then, if the region is in the small object area it calls function `int is_small_object (void*, struct gc_region)`. This function first checks whether the block containing the pointer is in use, and if yes, it goes from object to object in the block, and checks if the pointer points to the beginning of an object. If yes, it is a pointer, if not, it is ignored by the garbage collector.

If the region is in a large object area, the function `int large_set_exists(void*)` is called. This function searches through the large objects hash table in order to find if the object is there or not. If it is there, then it could indeed be a pointer, and the object is marked. Otherwise, it is ignored.

Variable/Function/Macro	Description	Source file
`gc_root_bundle`	struct gc_root_bundle: `int size` number of objects in bundle; `struct gc_root_bundle *next` next node in root bundle list; `struct` { `void *object` the object itself; `void (*walk) (void*)` the walking function for the object } `roots[ROOTBUNDLESIZE]`	gc.c
`void **mark_stack`	the mark stack	gc.c
`void gc_attach (void *root, gc_type *type)`	attach a root object to the garbage collector	gc.c
`void walk_roots (void)`	mark root objects	gc.c
`void walk_mark_stack (void)`	mark reachable objects	gc.c
`inline void mark (void*)`	mark an object	gc.c
`WALK(p)`	`(((Hjava_lang_Object*)(p))->dtable-> class->walk)(p)`	gc.c
`void makeWalkFunc (Hjava_lang_Class *class)`	create a walk function for a class	gc.c
`void handle_stack_overflow (void)`	handle mark stack overflow	gc.c
`void walk_small_objects (void)`	walk marked objects in small object area	gc.c
`void walk_large_objects (void)`	walk marked objects in large object area	gc.c

Table 4.7: Important variables and functions used for marking phase

Function	Description	Source file
`int is_object (void*)`	check if the reference is valid	gc.c
`int is_small_object (void*, struct gc_region*)`	check whether a pointer points to an object in the small object area	gc.c
`int large_set_exists (void *p)`	check if pointer is in large object table	gc.c

Table 4.8: Important functions for finding pointers

If the region is in the explicitly managed heap, or there are no regions containing the pointer, then it is ignored by the garbage collector.

Functions used for finding pointers are listed in Table 4.8.

Sweeping phase

After the marking phase is completed, the sweeping phase takes place, freeing the unused memory (see Table 4.9). Sweeping is done separately for small and large object area.

For small area two possible sweeping algorithms are used : selective sweeping (`void sweep_small_selective(void **, int)`), used when heap occupancy is low, and traditional sweeping (`void sweep_small_normal()`) used when the number of live objects is bigger than a certain threshold. The algorithms are chosen at run-time, thus improving garbage collection time, according to the authors. Selective sweeping takes as an input a set of live objects, sorts them by increasing addresses and frees space between them in constant time. Free memory is put back into free lists of blocks or chunks (depending on its size) using the function `void insert_freemem (struct small_chunk_table*, struct gc_small_block**, void*,void*)`. In the case of traditional sweeping, objects are visited one by one, and if they are not marked, their space is reclaimed. While sweeping, contiguous free memory (pointed to by `*slack`) is coalesced. It is put on one of the free lists (chunks or blocks) according to its size. In fact, lists of free chunks are made by the sweeping phase. Chunks are put in the appropriate list using the function `void insert_freechunk (struct small_chunk_table, gc_head, size_t)`, and after the sweeping all the lists are merged into one in order of decreasing object sizes (function `struct gc_small_chunk* merge_freeindex(struct small_chunk_table)`) to be able to accommodate worst-fit allocation.

The large object area is swept by looking at each object in the large object hash table `large_objects`. If the object is not marked, it is freed using the function `void mem_free(struct gc_heap, void *)` from manual memory manager. For both the small and large object areas, marked objects are unmarked so that the marking process in the next garbage collection works properly.

Variable/Function/Macro	Description	Source file
`gc_head *slack`	points to the beginning of the free area between two marked small objects	gc.c
`void sweep_small_selective (void **, int)`	selectively sweep the small object area	gc.c
`void sweep_small_normal()`	sweep the small object area traditionally	gc.c
`void insert_freemem (struct small_chunk_table*, struct gc_small_block**, void*,void*)`	insert a range of free memory into the free chunk and blocks list	gc.c
`void insert_freechunk (struct small_chunk_table, gc_head, size_t)`	insert a free chunk into the free chunk list index table	gc.c
`struct gc_small_chunk* merge_freeindex(struct small_chunk_table)`	merge the free chunk index table into a single list sorted in decreasing order of size	gc.c
`void mem_free(struct gc_heap, void *)`	manually free a memory chunk	gc.c

Table 4.9: Important variables and functions used for sweeping phase

Variable/Function/Macro	Description	Source file
`finalize_node`	struct finalize_node `void *object` — object with the finalizer `void (*final)(void*)` — finalizer `struct finalize_node *next` — next node in linked list	gc.c
`struct finalize_node *has_final`	list of live objects that have finalizers	gc.c
`struct finalize_node *do_final`	list of dead objects that have finalizers	gc.c
`void walk_finals (void)`	mark objects with finalizers	gc.c
`void invoke_finalizer (Hjava_lang_Object *object, void (*final)(void*))`	invoke the finalizer for an object	gc.c

Table 4.10: Important variables and functions used for object finalizing

Finalizers

In Java, the finalizer of an object is executed, even if the object is unreachable. The execution of a finalizer can make that object and its children reachable again, so a special care is needed for objects with finalizers. During the allocation, such objects are put in a linked list called `has_final`. After all the reachable objects are marked, the function `void walk_finals ()` removes the unmarked (dead) objects in that list and puts them in the `do_final` list. As the objects in a `do_final` list can be revived, they are put on the marking stack, and the marking (function `void walk_mark_stack ()`) is called again, in order to mark all the objects with finalizers and their children. The finalizer itself is invoked after the sweeping phase (with `void invoke_finalizer (void *object, void (*final)(void*))`), on the objects from the `do_final` list : objects that are dead, but have a finalizer.

Finalizers are executed in a separate thread, protected by `final_list_lock` mutex.

4.4 Conclusion

LaTTe's memory manager uses a rather elaborate allocator and garbage collector. Memory allocator uses different algorithms for different object types/sizes : pointer increments and free lists for small objects, free lists as hash tables for large objects, and free lists for different sizes for both explicitly managed heap and large objects area. Garbage collector uses mark and sweep algorithm, and different methods for sweeping of small objects (selective and traditional sweeping), based on heap occupancy.

The code of LaTTe memory manager is well written, commented and localized (files gc.h and gc.c cover it all, although they use some other parts of the JVM : like general types, exception and error handling and threads).

In the next chapter we will discuss possible reasons for the efficiency of LaTTe's memory manager, some improvements that can be introduced, as well as different algorithms that could be implemented, in order to test them and possibly improve the present algorithm.

Chapter 5

Discussion and Possible Improvements of LaTTe's Garbage Collector

5.1 Introduction

In this chapter, we discuss possible improvements to the existing LaTTe's garbage collector, as well as algorithms that could replace the existing one (while using some of its data structures). This is more a discussion on the algorithms, and not the exact implementation. It would have been nice to have an actual implementation, but it is unfortunately (because of lack of time) beyond the scope of this work.

5.2 LaTTe's Algorithm

5.2.1 Reasons for Its Efficiency

LaTTe's garbage collector performs much better than other mark-sweep collectors (see Section 3.6). As we did not enter into details of other mark-sweep implementations, we cannot compare them, but we will notice few things that possibly make LaTTe's more efficient.

One of the biggest issues of mark-sweep garbage collection is fragmentation. Although our benchmarks were insufficient to prove whether the heap expansion is due to the fragmentation or to real lack of memory, we think that the fragmentation may not be an issue here after all.

LaTTe has a well defined allocator, using the segregated lists and size classes for large and fixed objects (which is a sort of best fit), and pointer increments and lazy worst fit for small objects. Chung [60] indeed shows that pointer increments with fits as a backup (lazy fits) can give better performance than conventional fits. Although worst fit tends to add to fragmentation, LaTTe's garbage collector reconstructs from the scratch the free list after every sweep phase, therefore eliminating the fragmentation introduced in the previous allocation.

Besides, Johnstone [61] found that objects allocated at the same time tend to die at the same time, which explains a good behavior of the lazy worst fit : objects allocated in one free chunk using pointer increments, and just if there is no space in a free chunk, a new chunk is taken from the free list (using the worst fit); in average, all the objects being allocated in one free chunk would die at the same time, thus returning the whole chunk to the free list.

As for the marking phase, LaTTe uses customized marking function for every class. LaTTe's authors show [59] that it gives much better performance than using generic marking functions.

We already mentioned (see Section 4.3.2) the LaTTe's selective sweeping, which makes sweeping phase faster by visiting only live objects, and not the whole heap, when the heap occupancy is low. For more details see [9].

Finally, LaTTe uses the following heap expansion heuristics : the heap is expanded only when the amount of live objects exceeds the amount of objects allocated (the amount expanded being the difference between the two quantities). LaTTe's team compared this heuristics with "expand when needed" one and concluded that garbage collection time can significantly be improved using their heuristic.

5.2.2 Possible Improvements of LaTTe's Garbage Collector

Many possible improvements have already been mentioned by LaTTe's team [59]. We will mention just the ones that we noticed, and which concern the algorithm itself and not the actual implementation.

LaTTe uses lazy worst fit for small object allocation. It is surely the most efficient one, but it leads to fragmentation faster. It would be interesting to compare the performance of first or best fit with the present one.

Pointers to large objects are held in a hash table. During the marking phase, the hash table is searched, and every object in the heap marked, by changing a mark bit in its header. It would probably be better to hold mark bit in the hash table, so that there is no unnecessary access to the objects themselves (thus improving cache behavior).

5.3 Choice of the New Algorithm

LaTTe's collector proved to be very efficient and it seems hard to make a faster collector. At first sight, fragmentation does not seem to be an issue, but it has to be thoroughly tested and preferably on long-running applications, to be able to say that the fragmentation problem is not present. Even if it is not the case, it would still be interesting to use LaTTe to compare its mark-sweep with some compacting algorithms. The alternatives are mark-compact, copying and generational collector.

5.3.1 Compacting Garbage Collection Algorithms

Generational collection (Section 2.4) seems to be the most robust one, but the write barrier demands global changes in the JVM code, which makes the implementation more difficult. Mark-compact collection (Section 2.3.4) would be interesting, because the marking phase is already implemented in LaTTe's collector, so we would have to implement only the compacting phase, but it lacks performance due to number of passes that mark-compact algorithms do (2 to 3). Copying collection (Section 2.3.5) is mostly used in generational collectors, so implementing a copying collector would be the first step to it (first generations are frequently implemented as a copying collector). On the other hand, not much of an existing code can be reused for the copying collector, so we would have to make it from scratches.

5.3.2 Copying Garbage Collection and LaTTe

Conservatism and block size

LaTTe uses a partially conservative collector. That means that the collector knows the type of the heap objects, but not the type of the local variables and stack operands. So, if we were to use a copying collector, we would have to pay attention not to move objects pointed to by stack objects, as we are not certain of their type. A copying collector that is partially conservative is Bartlett's collector (Section 2.6.3). It leaves in place the objects referenced by stack objects, and copies all the others.

Bartlett's collector does not divide the heap into two semi-spaces, as an ordinary copying collector does. Instead, it divides the heap into blocks (or "pages", the term used by Bartlett, which should not be mistaken for virtual memory page) of 512 bytes each. Every block has a space identifier : a small field which identifies the space to which an object belongs (fromspace or tospace)[1]. During the collection, objects are copied from fromspace to tospace, as in standard copying collection. Only for the objects that cannot be copied, the "copying" is done by changing the space identifier of the block to which they belong. Unfortunately, this way all the objects on that block are retained regardless of whether they are live or not.

This leads to an important question of the block (page) size : as the block size gets bigger, more garbage is retained. On the other hand, smaller blocks make the allocation more difficult. The size of 512 bytes is chosen in regard to these two issues. Bartlett's collector is initially made for Lisp, which has small, equally sized cells and this choice of size seems to work well for Lisp. Java objects are much bigger than Lisp's cells, so this block size seems to be too small for Java. LaTTe's block size is 4K, and the limit between small and large objects is 1K (objects larger than 1K are considered large). The actual size of the block should be determined by experimenting with several sizes.

Cost of copying large objects

The other problem are large objects : the cost of copying large objects is bigger than that of smaller ones, which can be obtrusive. One solution is not to copy them, but to keep them in place, changing just the space identifier (just like it does for objects

[1]This identifier can hold the age information, thus making it easy to promote Bartlett's copying collector to generational

that are treated conservatively). This means that all other objects in that block would be kept alive, although they could be dead. Another solution is to keep large objects in a separate large objects space (which already exists in LaTTe, for objects bigger than 1K) and to collect that space differently : mark-sweep would be a good choice, because it is already present in LaTTe, but occasional mark-compact collection would be necessary if we want to completely avoid fragmentation.

Heap organization and pointer finding

LaTTe divides the heap in three parts : explicitly managed, large object and small object heap. In view of previously mentioned problem of copying large objects, our mostly copying collector for LaTTe could use the same scheme. Fixed objects can be treated in the same way, being allocated in the explicitly managed part and not garbage collected. Large objects can be allocated in large object heap and occasionally garbage collected, either by mark-sweep or mark-compact collection. Small objects, whose size should be determined, would be allocated in small object heap, and garbage collected by mostly copying collector.

In fact, we could have a pool of free blocks and allocate all the objects from them, but paying attention on a space identifier, which can be different for fixed, large and small objects. So if an object is the first to be allocated in a certain block, at that moment the block identifier would be set and the block put on one of three lists (for three types of objects). The objects would then be allocated from the appropriate block, by incrementing the pointer. If the allocation fails, a new block would be issued from the free pool, its identifier changed accordingly (free, large or small area) and the allocation continued.

The main problem with this kind of heap organization is the same as with the generational collection : as small and large object heap would not be collected at the same time, pointers from one to another heap must be updated with each copying collection so some kind of treatment for this case should exist. The easiest way is to scan all large objects for pointers to small objects, and to update their values : it means that both large and small object heap would be collected at the same time. It would be faster if large objects were divided into header and body, and if by scanning a separated list of headers, the pointer values could be updated. Or, we could maintain a remembered set : a set of all pointers from large to small objects. This would put an overhead on each store and would require changes in existing object formats, which is rather complicated.

Breadth-first or depth-first traversal ?

Bartlett's collector originally uses breadth-first traversal (see Cheney's algorithm, section 2.3.5) while scanning live objects : after "moving" the objects pointed by roots (in fact, just changing the space identifier of the corresponding block), these objects are scanned for pointers and the pointed objects are copied to tospace. Tospace objects are further scanned for pointers to the objects that are not yet copied, and the process is repeated until all live objects are copied. Cheney's algorithm is elegant, but it is found that a breadth-first traversal yields worse locality of references than a depth first. The locality is important for performance, because a good locality avoids frequent page eviction. Breadth-first traversal is typically implemented using a mark queue (FIFO) and depth traversal using a mark stack (LIFO).

5.3.3 Other Algorithms

Having a mark-sweep algorithm, it should possibly be easier to the Java virtual machine programmer to implement incremental mark-sweep collection. The changes to the overall virtual machine would be required, because of the write barrier that has to be implemented to register each pointer write, so we did not consider it. We suppose that LaTTe's team has it in mind for future work.

5.4 Conclusion

LaTTe has an efficient garbage collector. It is a well implemented mark-sweep algorithm, with selective sweeping and a good allocation policy. But, as it is not a copying collector, fragmentation still can be a problem.

It would be interesting to compare the performance of a copying collector with LaTTe's mark-sweep, and see if some improvements are possible. Not only for the fragmentation issue, but also because copying collector is the first step to generational one, which should be more efficient. The only copying collector that can be used in a semi-conservative way is the Bartlett's collector. We discussed the issues concerning this collector : block size, copying of large objects, traversing algorithm, and proposed alternatives.

The next step would be to implement those alternative algorithms, and to compare their performance to that of LaTTe's present collector.

Chapter 6

Conclusion

Memory management is a highly complex issue, having been under development for almost forty years, but still being rather mysterious. The allocation and deallocation algorithms are fairly known, but their combination, implementation and sophisticated details may influence greatly their performance. Java raised again the question of automatic memory management (garbage collection) efficiency by being among the most used language today, having a high allocation rate and having a mandatory garbage collection system.

In order to get some insight on memory management (with an accent on Java), we first tried to understand the basic allocation and garbage collection algorithms. Then we tested garbage collectors in six Java virtual machines, known for having implemented different garbage collection algorithms. We showed that a well-implemented simple garbage collection algorithm (LaTTe's) can have the same performance as a highly sophisticated, complex set of algorithms (Hotspot). Our benchmarks were rather simple, and more thorough analysis should be performed (using more applications, and more sophisticated profiling methods, such as having heap snapshots by instrumenting each JVM), but our results correspond to the results found by LaTTe's team, and we tried to give some possible explanation for this unexpected efficiency (which is much better than the same algorithm implemented in other JVMs).

This could have been done only by the analysis of the source code of LaTTe's memory manager. By reverse engineering of LaTTe's memory management code, we were able to understand in details the algorithms used, to find points for possible improvements, and to discuss other algorithms that could be implemented instead of the present one. That would in fact be possible future work : to implement several memory allocation and garbage collection algorithms in LaTTe and to test them. Some of that work has already been done by LaTTe's team, and we assume that they will further improve it, possibly going toward incremental collection (to satisfy real-time applications).

Bibliography

[1] B. Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[2] Latte web site. `http://latte.snu.ac.kr/`.

[3] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, Sept 1995.

[4] R. E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, 1996.

[5] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, Saint-Malo (France), 1992. Springer-Verlag.

[6] L. P. Deutsch and D. G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[7] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[8] R. John M. Hughes. A semi-incremental garbage collection algorithm. *Software Practice and Experience*, 12(11):1081–1084, November 1982.

[9] Y. C. Chung, S. Moon, K. Ebcioglu, and D. Sahlin. Reducing sweep time for a nearly empty heap. In *27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, Boston, MA, 2000. ACM Press.

[10] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[11] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 23(11), pages 1–17, New York, NY, 1988. ACM Press.

[12] R. Hudson and A. Diwan. Adaptive garbage collection for Modula-3 and Smalltalk. In *ECOOP/OOPSLA '90 Workshop on Garbage Collection*, 1990.

[13] A. L. Hosking and R. L. Hudson. Remembered sets can also play cards. In *OOPSLA 1993 Workshop on Memory Management and Garbage Collection*, 1993.

[14] B. Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23:733–756, 1993.

[15] H. 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.

[16] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.

[17] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[18] G. L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[19] H-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[20] J. F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, 1988.

[21] M. Campione and K. Walrath. *The Java Tutorial: Object-Oriented Programming for the Internet*. Addison-Wesley, Reading, 1996.

[22] O. Agesen. GC Mailing List Archive, March 1999. `http://lists.tunes.org/archives/gclist/1999-March/001550.html`.

[23] O. Agesen. Personal communication.

[24] Sun Microsystems Inc. The Java Hotspot Performance Engine Architecture : A White Paper About Sun's Second Generation Performance Technology, April 1999. `http://www.javasoft.com/products/hotspot/whitepaper.html#5`.

[25] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In *Proceedings of International Workshop on Memory Management, volume 637*, University of Massachusetts, USA, 1992.

[26] Sun's Java Hotspot web site. `http://java.sun.com/products/hotspot/index.html`.

[27] Kaffe web site. `http://www.kaffe.org/`.

[28] MachJ web site. `http://www.machj.com/more.htm`.

[29] TowerJ white paper. http://www.towerj.com/productsservices/whitepaperform.htm.

[30] HP JDK release notes for version 1.1.8.05. http://www.unix.hp.com/java/java1/jdk_jre/infolibrary/jdk_rnotes_11805.%html.

[31] HP ChaiVM web site. http://www.embedded.hp.com/products/platform/chaivm.html.

[32] Press release : HP introduces microchaiVM software for mobile devices, Feb 2001. http://www.hp.com/communications/news_events/press_releases/feb_2001/12%feb01a.html.

[33] IBM JDK web site. http://www-106.ibm.com/developerworks/java/jdk/.

[34] Hungry Programmers web site. http://www.hungry.com/.

[35] JOVE web site. http://www.instantiations.com/jove/product/thejovesystem.htm.

[36] M. Johnson. Jove optimizing native compiler for java technology. Technical report, Instantiations, Inc., 1999. http://www.instantiations.com/jove/jovereport.htm#TheJOVERuntimeEnvironment.

[37] JavaCard web site. http://java.sun.com/products/javacard/.

[38] J2ME Connected Limited Device Configuration (K Virtual Machine) download site. http://www.sun.com/software/communitysource/j2me/cldc/download.html.

[39] EmbeddedJava web site. http://www.javasoft.com/products/embeddedjava/.

[40] PersonalJava web site. http://www.javasoft.com/products/personaljava/.

[41] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real Time Specification for Java*. Addison-Wesley, 2000.

[42] PersonalJava Emulator web site. http://java.sun.com/products/personaljava/pj-emulation.html.

[43] PERC web site (Newmonics). http://www.newmonics.com.

[44] Charis web site. http://www.charis.com/.

[45] Jbed product line: Whitepaper. http://www.esmertec.com/p_whitepaper.html.

[46] CaffeineMark web site. http://www.pendragon-software.com/pendragon/cm3/info.html.

[47] The Benchmark Applet web site. `http://www.cts.com/browse/wholder/Doug/Benchmark/Benchmark.html`.

[48] UCSDBenchmarks web site. `http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html`.

[49] Java Grande Forum Benchmark Suite web site. `http://www.epcc.ed.ac.uk/javagrande/`.

[50] CUP web site. (http://www.cs.princeton.edu/ appel/modern/java/CUP/.

[51] JLex web site. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[52] Hyperprof web site. `http://www.physics.orst.edu/~bulatov/HyperProf/`.

[53] D. Stefanovic, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99)*, pages 370–381, Denver, Colorado, 1999.

[54] M. W. Hicks, J. T. Moore, and S. M. Nettles. The measured cost of copying garbage collection mechanisms. In *Proceedings of International Conference on Functional Programming*, Amsterdam, 1997.

[55] OptimizeIt web site. `http://www.optimizeit.com/`.

[56] JProbe web site. `http://www.sitraka.com/software/jprobe/`.

[57] Microprocessor Architecture and System Software Laboratory, School of Electrical Engineering, Seoul National University, web site. `http://altair.snu.ac.kr/`.

[58] VLIW (Very Long Instruction Word) at IBM Research, web site. `http://www.research.ibm.com/vliw/`.

[59] Y. Chung, J. Lee, S. Moon, and K. Ebcioglu. Memory management in the LaTTe Java virtual machine, 2000. In preparation.

[60] Y. Chung and S. Moon. Memory allocation with lazy fits. In submission, 2000.

[61] M. S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, The University of Texas at Austin, December 1997.