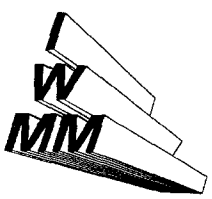
Lecture Notes in Computer Science 637
Edited by G. Goos and J. Hartmanis

Y. Bekkers J. Cohen (Eds.)

Memory Management

International Workshop IWMM 92
St. Malo, France, September 17-19, 1992
Proceedings



Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Volume Editors

Yves Bekkers
IRISA, Campus de Beaulieu
F-35042 Rennes, France

Jacques Cohen
Mitchum School of Computer Science, Ford Hall
Brandeis University, Waltham, MA 02254, USA

CR Subject Classification (1991): D.1, D.3-4, B.3, E.2

ISBN 3-540-55940-X Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-55940-X Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author/editor
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

Storage reclamation became a necessity when the Lisp function *cons* was originally conceived[1]. That statement is simply a computer-oriented version of the broader precept: Recycling becomes unavoidable when usable resources are depleted. Both statements succinctly explain the nature of the topics discussed in the *International Workshop on Memory Management* (IWMM) that took place in Saint-Malo, France, in September 1992. This volume assembles the refereed technical papers which were presented during the workshop.

The earlier programming languages (such as Fortran) were designed so that the size of the storage required for the execution of a program was known at compile time. Subsequent languages (such as Algol 60) were implemented using a *stack* as a principal data-structure which is managed dynamically: information pushed onto a stack uses memory space which can be later released by popping.

With the introduction of structures (also called records) in more recent programming languages, it became important to establish an additional run-time data structure: the *heap*, which is used to store data-cells containing pointers to other cells. The *stack-heap* arrangement has become practically universal in the implementation of programming languages. An important characteristic of the cells in the heap is that the data they contain can become "useless" since they are not pointed to by any other cells. Reclamation of the so-called "useless cells" can be performed in an *ad hoc* (manual) manner by having the programmer explicitly return those cells to the run-time system so that they can be reused. (In *ad hoc* reclamation the programmer has to exercise great caution not to return cells containing valuable data.) This is the case of languages like Pascal or C which provide primitive procedures for returning useless cells. In the case of languages such as Lisp and Prolog reclamation is done automatically using a run-time process called *garbage-collection* which detects useless cells and makes them available for future usage.

Practically all the papers in this volume deal with the various aspects of managing and reclaiming memory storage when using a *stack-heap* model. A peculiar problem of memory management strategies is the unpredictability of computations. The undecidability of the halting problem implies that, in general, it is impossible to foresee how many cells will be needed in performing complex computations.

There are basically two approaches for performing storage reclamation: one is *incremental*, i.e., the implementor chooses to blend the task of collecting with that of actual computation; the other is what we like to call the *mañana* method - wait until the entire memory is exhausted to trigger the time-consuming operation of recognizing useless cells and making them available for future usage. A correct reclamation should ensure the following properties:

- *No used cell will be (erroneously) reclaimed.*
- *All useless cells will be reclaimed.*

Violating the first property is bound to have tragic consequences. A violation of the second may not be disastrous, but could lead to a premature halting of the execution due to the lack of memory. As a matter of fact, *conservative* collectors have been proposed to trade a (small) percentage of unreclaimed useless cells for a speedup of the collection process.

An important step in the collection is the identification of useless cells. This can be achieved by *marking* all the useful cells and *sweeping* the entire memory to collect useless

[1] The reader is referred to the chapter on the *History of Lisp*, by John McCarthy, which appeared in History of Programming Languages, edited by Richard L. Wexelblat, Academic Press, 1981, pp 173-183.

(unmarked) cells. This process is known as *mark-and-sweep*. Another manner of identifying useless cells is to keep *reference counts* which are constantly updated to indicate the number of pointers to a given cell. When this number becomes zero the cell is identified as useless. If the mark-and-sweep or the reference count techniques fail to locate any useless cells, the program being executed has to halt due to lack of storage. (A nasty situation may occur when successive collections succeed in reclaiming only a few cells. In such cases very little actual computation is performed between consecutive time-consuming collections.)

Compacting collectors are those which compact the useful information into a contiguous storage area. Such compacting requires that pointers be properly readjusted. Compacting becomes an important issue in paging systems (or in the case of hierarchical or virtual memories) since the compacted useful information is likely to result in fewer page faults, and therefore in increased performance.

An alternative method of garbage-collection which has drawn the attention of implementors in recent years is that of *copying*. In this case the useful cells are simply copied into a "new" area from the "old" one. These areas are called semi-spaces. When the space in the "new" area is exhausted, the "old" and "new" semi-spaces are swapped. Although this method requires twice the storage area needed by other methods, it can be performed incrementally, thus offering the possibility of *real-time* garbage-collection, in which the interruptions for collections are reasonably short.

The so-called *generational* garbage-collection is based on the experimental fact that certain cells remain used during substantial periods of the execution of a program, whereas others become useless shortly after they are generated. In these cases the reclaiming strategy consists of bypassing the costly redundant identification of "old generation" cells.

With the advent of *distributed* and *parallel* computers reclamation becomes considerably more complex. The choice of storage management strategy is, of course, dependent on the various types of existing architectures. One should distinguish the cases of:

1. Distributed computers communicating via a network,
2. Parallel shared-memory (MIMD) computers, and
3. Massively parallel (SIMD) computers.

In the case of distributed reclamation it is important that collectors be fault tolerant: a failure of one or more processors should not result in loss of information. The term *on-the-fly* garbage-collection is (usually) applicable to parallel shared-memory machines in which one or more processors are dedicated exclusively to collecting while others, called *mutators*, are responsible for performing useful computations which in turn may generate useless cells that have to be reclaimed.

Some features of storage management are *language-dependent*. Presently, one can distinguish three major paradigms in programming language design: *functional*, *logic*, and *object-oriented*. Although functional languages, like Lisp, were the first to incorporate garbage-collection in their design, both logic and object-oriented language implementors followed suit. Certain languages have features that enable their implementors to take advantage of known properties of data in the stack or in the heap so as to reduce the execution time needed for collection and/or to reclaim as many useless cells as possible.

In the preceding paragraphs we have briefly defined the terms: *mark-and-sweep*, *reference count*, *compacting*, *copying*, *incremental*, *generational*, *conservative*, *distributed*, *parallel*, *on-the-fly*, *real-time*, and *language-dependent features*. These terms should serve to guide the reader through the various papers presented in this volume.

We suggest that non-specialists start by reading the three survey papers. The first provides a general overview of the recent developments in the field; the second specializes in distributed collection, and the third deals with storage management in processors for logic programs. The other chapters in this volume deal with the topics of distributed, parallel, and

incremental collections, collecting in functional, logic, and object-oriented languages, and collections using massively parallel computers. The final article in this volume is an invited paper by H. G. Baker in which he proposes a "reversible" Lisp-like language (i.e., capable of reversing computations) and discusses the problems of designing suitable garbage-collectors for that language.

We wish to thank the referees for their careful evaluation of the submitted papers, and for the suggestions they provided to the authors for improving the quality of the presentation. Finally, it is fair to state that, even with technological advances, there will always be limited memory resources, especially those of very fast access. These memories will likely remain costlier than those with slower access. Therefore many of the solutions proposed at the IWMM are likely to remain valid for years to come.

July 1992

Yves Bekkers
Jacques Cohen

Program Committee

Chair	
Jacques Cohen	Brandeis University, Waltham, MA, USA
Members	
Joel F. Bartlett	DEC, Palo Alto, CA, USA
Yves Bekkers	INRIA-IRISA, Rennes, France
Hans-Jurgen Boehm	Xerox Corporation, Palo Alto, CA, USA
Maurice Bruynooghe	Katholieke Universiteit, Leuven, Belgium
Bernard Lang	INRIA, Le Chesnay, France
David A. Moon	Apple Computer, Cambridge, MA, USA
Christian Queinnec	Ecole Polytechnique, Palaiseau, France
Dan Sahlin	SICS, Kista, Sweden
Taiichi Yuasa	Toyohashi Univ. of Tech., Toyohashi, Japan

We thank all the people who helped the program committee in the refereering process, some of whom are listed below: K. Ali, M. Banâtre, P. Brand, A. Callebou, P. Fradet, S. Jansson, P. Magnusson, A. Mariën, R. Moolenaar, A. Mulkers, O. Ridoux, A. Saulsbury, T. Sjöland, L. Ungaro, P. Weemeeuw.

Workshop Coordinator

Yves Bekkers — INRIA-IRISA, Rennes, France

Sponsored by

INRIA
University of Rennes I
CNRS-GRECO Programmation

In cooperation with

ACM SIGPLAN

Table of Contents

Surveys

Distributed Systems I

Distributed Systems II

Parallelism I

Functional languages

Logic Programming Languages I

Object Oriented Languages

Logic Programming Languages II

Incremental

Improving Locality

Parallelism II

Massively Parallel Architectures

Invited Speaker