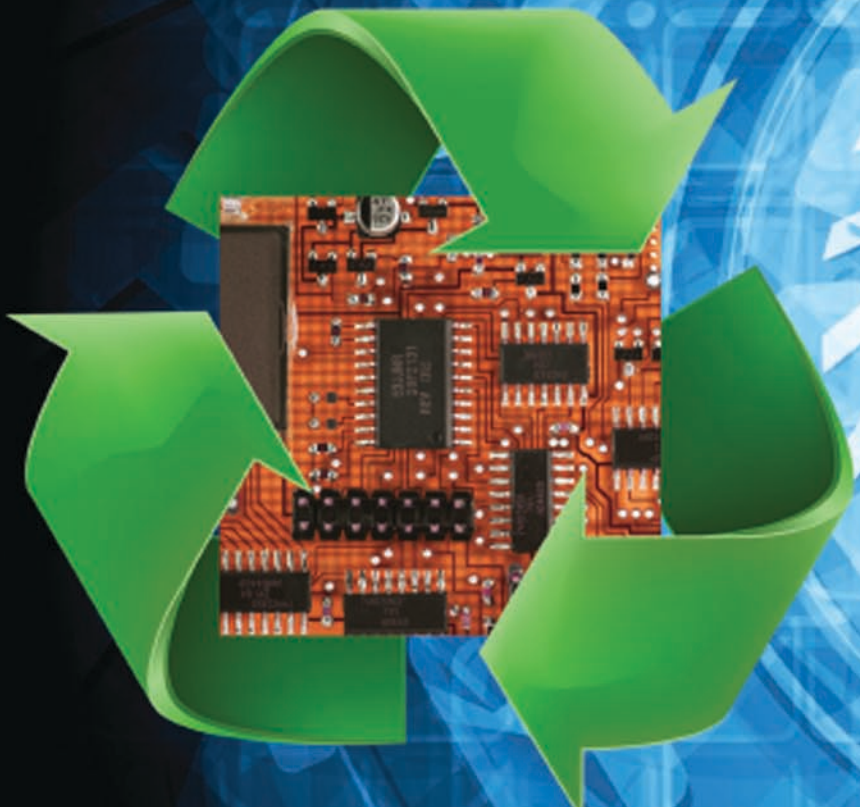
THE GARBAGE COLLECTION HANDBOOK

The Art of Automatic Memory Management



Richard Jones
Antony Hosking
Eliot Moss

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

THE GARBAGE COLLECTION HANDBOOK

The Art of Automatic Memory Management

Chapman & Hall/CRC

Applied Algorithms and Data Structures Series

Series Editor

Samir Khuller
University of Maryland

Aims and Scopes

The design and analysis of algorithms and data structures form the foundation of computer science. As current algorithms and data structures are improved and new methods are introduced, it becomes increasingly important to present the latest research and applications to professionals in the field.

This series aims to capture new developments and applications in the design and analysis of algorithms and data structures through the publication of a broad range of textbooks, reference works, and handbooks. The inclusion of concrete examples and applications is highly encouraged. The scope of the series includes, but is not limited to, titles in the areas of parallel algorithms, approximation algorithms, randomized algorithms, graph algorithms, search algorithms, machine learning algorithms, medical algorithms, data structures, graph structures, tree data structures, and other relevant topics that might be proposed by potential contributors.

Published Titles

A Practical Guide to Data Structures and Algorithms Using Java
Sally A. Goldman and Kenneth J. Goldman

Algorithms and Theory of Computation Handbook, Second Edition – Two Volume Set
Edited by Mikhail J. Atallah and Marina Blanton

Mathematical and Algorithmic Foundations of the Internet
Fabrizio Luccio and Linda Pagli, with Graham Steel

The Garbage Collection Handbook: The Art of Automatic Memory Management
Richard Jones, Antony Hosking, and Eliot Moss

THE GARBAGE COLLECTION HANDBOOK

The Art of Automatic Memory Management

Richard Jones
Antony Hosking
Eliot Moss

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business

A CHAPMAN & HALL BOOK

The cover image logo concept was created by Richard Jones, and the rights to the logo belong solely to him.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
Version Date: 20160606

International Standard Book Number: 978-1-4200-8279-1 (Hardback)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To
Robbie, Helen, Kate and William
Mandi, Ben, Matt, Jory and K
Hannah, Natalie and Casandra

Contents

List of Algorithms

List of Figures

List of Tables

Preface

Happy anniversary! As we near completion of this book it is also the 50th anniversary of the first papers on automatic dynamic memory management, or *garbage collection*, written by McCarthy and Collins in 1960. Garbage collection was born in the Lisp programming language. By a curious coincidence, we started writing on the tenth anniversary of the first *International Symposium on Memory Management*, held in October 1998, almost exactly 40 years after the implementation of Lisp started in 1958. McCarthy [1978] recollects that the first online demonstration was to an MIT Industrial Liaison Symposium. It was important to make a good impression but unfortunately, mid-way through the demonstration, the IBM 704[1] exhausted (all of!) its 32k words of memory — McCarthy's team had omitted to refresh the Lisp core image from a previous rehearsal — and its Flexowriter printed, at ten characters per second,

```
THE GARBAGE COLLECTOR HAS BEEN CALLED. SOME INTERESTING
STATISTICS ARE AS FOLLOWS:
```

and so on at great length, taking all the time remaining for the demonstration. McCarthy and the audience collapsed in laughter. Fifty years on, garbage collection is no joke but an essential component of modern programming language implementations. Indeed, Visual Basic (introduced in 1991) is probably the only widely used language developed since 1990 not to adopt automatic memory management, but even its modern incarnation, VB.NET (2002), relies on the garbage collector in Microsoft's Common Language Runtime.

The advantages that garbage collected languages offer to software development are legion. It eliminates whole classes of bugs, such as attempting to follow dangling pointers that still refer to memory that has been reclaimed or worse, reused in another context. It is no longer possible to free memory that has already been freed. It reduces the chances of programs leaking memory, although it cannot cure all errors of this kind. It greatly simplifies the construction and use of concurrent data structures [Herlihy and Shavit, 2008]. Above all, the abstraction offered by garbage collection provides for better software engineering practice. It simplifies user interfaces and leads to code that is easier to understand and to maintain, and hence more reliable. By removing memory management worries from interfaces, it leads to code that is easier to reuse.

The memory management field has developed at an ever increasing rate in recent years, in terms of both software and hardware. In 1996, a typical Intel Pentium processor had a clock speed of 120 MHz although high-end workstations based on Digital's Alpha chips could run as fast as 266 MHz! Today's top-end processors run at over 3 GHz and multicore chips are ubiquitous. The size of main memory deployed has similarly increased nearly 1000-fold, from a few megabytes to four gigabytes being common in desktop machines

[1] The IBM 704's legacy to the Lisp world includes the terms *car* and *cdr*. The 704's 36-bit words included two 15-bit parts, the address and decrement parts. Lisp's list or *cons cells* stored pointers in these two parts. The head of the list, the car, could be obtained using the 704's `car` 'Contents of the Address part of Register' instruction, and the tail, the cdr, with its `cdr` 'Contents of the Decrement part of Register' instruction.

today. Nevertheless, the advances made in the performance of DRAM memory continue to lag well behind those of processors. At that time, we wrote that we did not argue that "garbage collection is a panacea for all memory management problems," and in particular pointed out that "the problem of garbage collection for hard real-time programming [where deadlines must be met without fail] has yet to be solved" [Jones, 1996]. Yet today, hard real-time collectors have moved out of the research laboratory and into commercially deployed systems. Nevertheless, although many problems have been solved by modern garbage collector implementations, new hardware, new environments and new applications continue to throw up new research challenges for memory management.

The audience

In this book, we have tried to bring together the wealth of experience gathered by automatic memory management researchers and developers over the past fifty years. The literature is huge — our online bibliography contains 2,500 entries at the time of writing. We discuss and compare the most important approaches and state-of-the-art techniques in a single, accessible framework. We have taken care to present algorithms and concepts using a consistent style and terminology. These are described in detail, often with pseudocode and illustrations. Where it is critical to performance, we pay attention to low level details, such as the choice of primitive operations for synchronisation or how hardware components such as caches influence algorithm design.

In particular, we address the new challenges presented to garbage collection by advances in hardware and software over the last decade or so. The gap in performance between processors and memory has by and large continued to widen. Processor clock speeds have increased, more and more cores are being placed on each die and configurations with multiple processor modules are common. This book focuses strongly on the consequences of these changes for designers and implementers of high performance garbage collectors. Their algorithms must take locality into account since cache performance is critical. Increasing numbers of application programs are multithreaded and run on multicore processors. Memory managers must be designed to avoid becoming a sequential bottleneck. On the other hand, the garbage collector itself should be designed to take advantage of the parallelism provided by new hardware. In Jones [1996], we did not consider at all how we might run multiple collector threads in parallel. We devoted but a single chapter to incremental and concurrent collection, which seemed exotic then.

We are sensitive throughout this book to the opportunities and limitations provided by modern hardware. We address locality issues throughout. From the outset, we assume that application programs may be multithreaded. Although we cover many of the more simple and traditional algorithms, we also devote nearly half of the book to discussing parallel, incremental, concurrent and real-time garbage collection.

We hope that this survey will help postgraduate students, researchers and developers who are interested in the implementation of programming languages. The book should also be useful to undergraduate students taking advanced courses in programming languages, compiler construction, software engineering or operating systems. Furthermore, we hope that it will give professional programmers better insight into the issues that the garbage collector faces and how different collectors work and that, armed with this knowledge, they will be better able to select and configure the choice of garbage collectors that many languages offer. The almost universal adoption of garbage collection by modern programming languages makes a thorough understanding of this topic essential for any programmer.

Structure of the book

Chapter 1 starts by considering why automatic storage reclamation is desirable, and briefly introduces the ways in which different garbage collection strategies can be compared. It ends with a description of the abstractions and pseudocode notation used throughout the rest of the book.

The next four chapters discuss the classical garbage collection building blocks in detail. We look at mark-sweep, mark-compact and copying garbage collection, followed by reference counting. These strategies are covered in depth, with particular focus on their implementation on modern hardware. Readers looking for a gentler introduction might also consult our earlier book *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Richard Jones and Rafael Lins, Wiley, 1996. The next chapter compares the strategies and algorithms covered in Chapters 2 to 5 in depth, assessing their strengths, weaknesses and applicability to different contexts.

How storage is reclaimed depends on how it is allocated. Chapter 7 considers different techniques for allocating memory and examines the extent to which automatic garbage collection leads to allocator policies that are different to those of explicit `malloc/free` memory management.

The first seven chapters make the implicit assumption that all objects in the heap are managed in the same way. However, there are many reasons why that would be a poor design. Chapters 8 to 10 consider why we might want to partition the heap into different spaces, and how we might manage those spaces. We look at generational garbage collection, one of the most successful strategies for managing objects, how to handle large objects and many other partitioned schemes.

The interface with the rest of the run-time system is one of the trickiest aspects of building a collector.[2] We devote Chapter 11 to the run-time interface, including finding pointers, safe points at which to collect, and read and write barriers, and Chapter 12 to language-specific concerns such as finalisation and weak references.

Next we turn our attention to concurrency. We set the scene in Chapter 13 by examining what modern hardware presents to the garbage collection implementer, and looking at algorithms for synchronisation, progress, termination and consensus. In Chapter 14 we see how we can execute multiple collector threads in parallel while all the application threads are halted. In the next four chapters we consider a wide range of concurrent collectors, in which we relax this 'stop-the-world' requirement in order to allow collection to take place with only the briefest, if any, interruptions to the user program. Finally, Chapter 19 takes this to its most challenging extreme, garbage collection for hard real-time systems.

At the end of each chapter, we offer a summary of issues to consider. These are intended to provoke the reader into asking what requirements their system has and how they can be met. What questions need to be answered about the behaviour of the client program, their operating system or the underlying hardware? These summaries are not intended as a substitute for reading the chapter. Above all, they are not intended as canned solutions, but we hope that they will provide a focus for further analysis.

Finally, what is missing from the book? We have only considered *automatic* techniques for memory management embedded in the run-time system. Thus, even when a language specification mandates garbage collection, we have not discussed in much depth other mechanisms for memory management that it may also support. The most obvious example is the use of 'regions' [Tofte and Talpin, 1994], most prominently used in the Real-Time Specification for Java. We pay attention only briefly to questions of region inferencing or stack allocation and very little at all to other compile-time analyses intended to replace, or

[2] And one that we passed on in Jones [1996]!

at least assist, garbage collection. Neither do we address how best to use techniques such as reference counting in the client program, although this is popular in languages like C++. Finally, the last decade has seen little new research on distributed garbage collection. In many ways, this is a shame since we expect lessons learnt in that field also to be useful to those developing collectors for the next generation of machines with heterogeneous collections of highly non-uniform memory architectures. Nevertheless, we do not discuss distributed garbage collection here.

e-book

The book is available either in print or as an e-book. The e-book has a number of enhancements over the print version. In particular, it is heavily hyperlinked. There are clickable links from every reference to its target (chapter, section, algorithm, figure, table and so on). Clicking on a citation will take the reader to its entry in the bibliography, many of which have a digital object identifier ('doi'), in turn a link to the original paper. Each entry in the bibliography also includes a list of pages from which it was cited; these are links back to the referring citations. Similarly, entries in the index contain links to the pages where the entry was mentioned. Finally, technical terms in the text have been linked to their entries in the glossary.

Online resources

The web page accompanying the book can be found at

`http://www.gchandbook.org`

It includes a number of resources including our comprehensive bibliography. The bibliography at the end of this book contains over 400 references. However, our comprehensive online database contains over 2500 garbage collection related publications. This database can be searched online or downloaded as BibTeX, PostScript or PDF. As well as details of the article, papers, books, theses and so on, the bibliography also contains abstracts for some entries and URLs or DOIs for most of the electronically available ones.

We continually strive to keep this bibliography up to date as a service to the community. Richard (R.E.Jones@kent.ac.uk) would be very grateful to receive further entries (or corrections).

Acknowledgements

We thank our many colleagues for their support for this new book. It is certain that without their encouragement (and pressure), this work would not have got off the ground. In particular, we thank Steve Blackburn, Hans Boehm, David Bacon, Cliff Click, David Detlefs, Daniel Frampton, Robin Garner, Barry Hayes, Laurence Hellyer, Maurice Herlihy, Martin Hirzel, Tomáš Kalibera, Doug Lea, Simon Marlow, Alan Mycroft, Cosmin Oancea, Erez Petrank, Fil Pizlo, Tony Printezis, John Reppy, David Siegwart, Gil Tene and Mario Wolczko, all of whom have answered our many questions or given us excellent feedback on early drafts. We also pay tribute to the many computer scientists who have worked on automatic memory management since 1958: without them there would be nothing to write about.

We are very grateful to Randi Cohen, our long-suffering editor at Taylor and Francis, for her support and patience. She has always been quick to offer help and slow to chide us for our tardiness. We also thank Elizabeth Haylett and the Society of Authors[3] for her service, which we recommend highly to other authors.

Richard Jones, Antony Hosking, Eliot Moss

Above all, I am grateful to Robbie. How she has borne the stress of another book, whose writing has yet again stretched well over the planned two years, I will never know. I owe you everything! I also doubt whether this book would have seen the light of day without the inexhaustible enthusiasm of my co-authors. Tony, Eliot, it has been a pleasure and an honour writing with knowledgeable and diligent colleagues.

Richard Jones

In the summer of 2002 Richard and I hatched plans to write a follow-up to his 1996 book. There had been lots of new work on GC in those six years, and it seemed there was demand for an update. Little did we know then that it would be another nine years before the current volume would appear. Richard, your patience is much appreciated. As conception turned into concrete planning, Eliot's offer to pitch in was gratefully accepted; without his sharing the load we would still be labouring anxiously. Much of the early planning and writing was carried out while I was on sabbatical with Richard in 2008, with funding from the United Kingdom's Engineering and Physical Sciences Research Council and the United States' National Science Foundation whose support we gratefully acknowledge. Mandi, without your encouragement and willingness to live out our own Canterbury tale this project would not have been possible.

Antony Hosking

[3] `http://www.societyofauthors.org`.

Thank you to my co-authors for inviting me into their project, already largely conceived and being proposed for publication. You were a pleasure to work with (as always), and tolerant of my sometimes idiosyncratic writing style. A formal thank you is also due the Royal Academy of Engineering, who supported my visit to the UK in November 2009, which greatly advanced the book. Other funding agencies supported the work indirectly by helping us attend conferences and meetings at which we could gain some face to face working time for the book as well. And most of all many thanks to my 'girls,' who endured my absences, physical and otherwise. Your support was essential and is deeply appreciated!

Eliot Moss

Authors

Richard Jones is Professor of Computer Systems at the School of Computing, University of Kent, Canterbury. He received a BA in Mathematics from Oxford University in 1976. He spent a few years teaching before returning to higher education at the University of Kent, where he has remained ever since, receiving an MSc in Computer Science in 1989. In 1998, he co-founded the International Symposium on Memory Management, of which he was the inaugural Programme Chair. He has published numerous papers on garbage collection, heap visualisation and electronic publishing, and he regularly sits on the programme committees of leading international conferences. He is a member of the Editorial Board of *Software Practice and Experience*. He was made an Honorary Fellow of the University of Glasgow in 2005 in recognition of his research and scholarship in dynamic memory management. He was named a Distinguished Scientist of the Association for Computing Machinery in 2006, and in 2014 a Fellow of the British Computer Society, a Fellow of the RSA and a member of AITO. He is married, with three children, and in his spare time he cycles and races Dart 18 catamarans.

Antony Hosking is Professor of Computer Science at the Australian National University, and Associate Professor of Computer Science at Purdue University, West Lafayette (on leave from 2015). He received a BSc in Mathematical Sciences from the University of Adelaide, Australia, in 1985, and an MSc in Computer Science from the University of Waikato, New Zealand, in 1987. He continued his graduate studies at the University of Massachusetts Amherst, receiving a PhD in Computer Science in 1995. His work is in the area of programming language design and implementation, with specific interests in database and persistent programming languages, object-oriented database systems, dynamic memory management, compiler optimisations, and architectural support for programming languages and applications. He was named a Distinguished Scientist of the Association for Computing Machinery in 2012, a member of AITO in 2013, and is a Member of the Institute of Electrical and Electronics Engineers. He regularly serves on programme and steering committees of major conferences, mostly focused on programming language design and implementation. He is married, with five children. When the opportunity arises, he most enjoys sitting somewhere behind the bowler's arm on the first day of any Test match at the Adelaide Oval.

Eliot Moss is a Professor in the Department of Computer Science at the University of Massachusetts Amherst. He received a BSEE in 1975, MSEE in 1978, and PhD in Computer Science in 1981, all from the Massachusetts Institute of Technology, Cambridge. After four years of military service, he joined the Computer Science faculty at the University of Massachusetts Amherst. He works in the area of programming languages and their implementation, and has built garbage collectors since 1978. In addition to his research on automatic memory management, he is known for his work on persistent programming

languages, virtual machine implementation, transactional programming and transactional memory. He worked with IBM researchers to license the Jikes RVM Java virtual machine for academic research, which eventually led to its release as an open source project. In 2007 he was named a Fellow of the Association for Computing Machinery and in 2009 a Fellow of the Institute of Electrical and Electronics Engineers. In 2012 he was co-recipient of the Edsger W. Dijkstra Prize in Distributed Computing for work on transactional memory. He served for four years as Secretary of the Association for Computing Machinery's Special Interest Group on Programming Languages, and served on many programme and steering committees of the significant venues related to his areas of research. Ordained a priest of the Episcopal Church in 2005, he leads a congregation in addition to his full-time academic position. He is married, with two children. He enjoys listening to recorded books and movie-going, and has been known to play the harp.

Chapter 1

Introduction

Developers are increasingly turning to managed languages and *run-time systems* for the many virtues they offer, from the increased security they bestow to code to the flexibility they provide by abstracting away from operating system and architecture. The benefits of *managed code* are widely accepted [Butters, 2007]. Because many services are provided by the virtual machine, programmers have less code to write. Code is safer if it is type-safe and if the run-time system verifies code as it is loaded, checks for resource access violations and the bounds of arrays and other collections, and manages memory automatically. Deployment costs are lower since it is easier to deploy applications to different platforms, even if the mantra 'write once, run anywhere' is over-optimistic. Consequently, programmers can spend a greater proportion of development time on the logic of their application.

Almost all modern programming languages make use of dynamic memory *allocation*. This allows objects to be allocated and deallocated even if their total size was not known at the time that the program was compiled, and if their lifetime may exceed that of the subroutine activation[1] that allocated them. A dynamically allocated object is stored in a *heap*, rather than on the *stack* (in the *activation record* or *stack frame* of the procedure that allocated it) or *statically* (whereby the name of an object is bound to a storage location known at compile or link time). Heap allocation is particularly important because it allows the programmer:

- to choose dynamically the size of new objects (thus avoiding program failure through exceeding hard-coded limits on arrays);
- to define and use recursive data structures such as lists, trees and maps;
- to return newly created objects to the parent procedure (allowing, for example, factory methods);
- to return a function as the result of another function (for example, *closures* or *suspensions* in functional languages).

Heap allocated objects are accessed through *references*. Typically, a *reference* is a *pointer* to the object (that is, the address in memory of the object). However, a reference may alternatively refer to an object only indirectly, for instance through a *handle* which in turn points to the object. Handles offer the advantage of allowing an object to be relocated (updating its handle) without having to change every reference to that object/handle throughout the program.

[1] We shall tend to use the terms *method*, *function*, *procedure* and *subroutine* interchangeably.

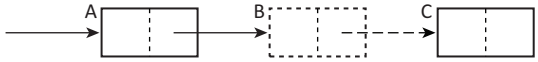


Figure 1.1: Premature deletion of an object may lead to errors. Here B has been freed. The live object A now contains a dangling pointer. The space occupied by C has leaked: C is not reachable but it cannot be freed.

1.1 Explicit deallocation

Any non-trivial program, running in a finite amount of memory, will need from time to time to recover the storage used by objects that are no longer needed by the computation. Memory used by heap objects can be reclaimed using *explicit deallocation* (for example, with C's `free` or C++'s `delete` operator) or automatically by the run-time system, using reference counting [Collins, 1960] or a tracing garbage collector [McCarthy, 1960]. Manual reclamation risks programming errors; these may arise in two ways.

Memory may be freed prematurely, while there are still references to it. Such a reference is called a *dangling pointer* (see Figure 1.1). If the program subsequently follows a dangling pointer, the result is unpredictable. The application programmer has no control over what happens to deallocated memory, so the run-time system may choose, among other options, to clear (fill with zeroes) the space used by the deleted object, to allocate a new object in that space or to return that memory to the operating system. The best that the programmer can hope for is that the program crashes immediately. However, it is more likely that it will continue for millions of cycles before crashing (making debugging difficult) or simply run to completion but produce incorrect results (which might not even be easy to detect). One way to detect dangling references is to use *fat pointers*. These can be used to hold the version number of their target as well as the pointer itself. Operations such as dereferencing must then check that the version number stored in the pointer matches that stored in the object. However, this approach is mostly restricted to use with debugging tools because of its overhead, and it is not completely reliable.[2]

The second kind of error is that the programmer may fail to free an object no longer required by the program, leading to a *memory leak*. In small programs, leaks may be benign but in large programs they are likely to lead either to substantial performance degradation (as the memory manager struggles to satisfy new allocation requests) or to failure (if the program runs out of memory). Often a single incorrect deallocation may lead to both dangling pointers and memory leaks (as in Figure 1.1).

Programming errors of this kind are particularly prevalent in the presence of sharing, when two or more subroutines may hold references to an object. This is even more problematic for concurrent programming when two or more threads may reference an object. With the increasing ubiquity of multicore processors, considerable effort has gone into the construction of libraries of data structures that are thread-safe. Algorithms that access these structures need to guard against a number of problems, including deadlock, livelock and ABA errors.[3] Automatic memory management eases the construction of concurrent algorithms significantly (for example, by eliminating certain ABA problems). Without this, programming solutions are much more complicated [Herlihy and Shavit, 2008].

The issue is more fundamental than simply being a matter of programmers needing to take more care. Difficulties of correct memory management are often inherent to the pro-

[2] Tools such as the `memcheck` leak detector used with the `valgrind` open source instrumentation framework (see `http://valgrind.org`) are more reliable, but even slower. There are also a number of commercially available programs for helping to debug memory issues.

[3] ABA error: a memory location is written (A), overwritten (B) and then overwritten again with the previous value A (see Chapter 13).

gramming problem in question.[4] More generally, safe deallocation of an object is complex because, as Wilson [1994] points out, "liveness is a *global* property", whereas the decision to call `free` on a variable is a local one.

So how do programmers cope in languages not supported by automatic dynamic memory management? Considerable effort has been invested in resolving this dilemma. The key advice has been to be consistent in the way that they manage the *ownership* of objects [Belotsky, 2003; Cline and Lomow, 1995]. Belotsky [2003] and others offer several possible strategies for C++. First, programmers should avoid heap allocation altogether, wherever possible. For example, objects can be allocated on the stack instead. When the objects' creating method returns, the popping of the stack will free these objects automatically. Secondly, programmers should pass and return objects by value, by copying the full contents of a parameter/result rather than by passing references. Clearly both of these approaches remove all allocation/deallocation errors but they do so at the cost of both increased memory pressure and the loss of sharing. In some circumstances it may be appropriate to use custom allocators, for example, that manage a pool of objects. At the end of a program phase, the entire pool can be freed as a whole.

C++ has seen several attempts to use special pointer classes and templates to improve memory management. These overload normal pointer operations in order to provide safe storage reclamation. However, such *smart pointers* have several limitations. The `auto_ptr` class template cannot be used with the Standard Template Library and will be deprecated in the expected next edition of the C++ standard [Boehm and Spertus, 2009].[5] It will be replaced by an improved `unique_ptr` that provides strict ownership semantics that allow the target object to be deleted when the unique pointer is. The standard will also include a reference counted `shared_ptr`,[6] but these also have limitations. Reference counted pointers are unable to manage self-referential (cyclic) data structures. Most smart pointers are provided as libraries, which restricts their applicability if efficiency is a concern. Possibly, they are most appropriately used to manage very large blocks, references to which are rarely assigned or passed, in which case they might be significantly cheaper than tracing collection. On the other hand, without the cooperation of the compiler and run-time system, reference counted pointers are not an efficient, general purpose solution to the management of small objects, especially if pointer manipulation is to be thread-safe.

The plethora of strategies for safe manual memory management throws up yet another problem. If it is essential for the programmer to manage object ownership consistently, which approach should she adopt? This is particularly problematic when using library code. Which approach does the library take? Do all the libraries used by the program use the same approach?

1.2 Automatic dynamic memory management

Automatic dynamic memory management resolves many of these issues. *Garbage collection* (GC) prevents dangling pointers being created: an object is reclaimed only when there is no pointer to it from a reachable object. Conversely, in principle all garbage is guaranteed to be freed — any object that is unreachable will eventually be reclaimed by the collector — with two caveats. The first is that *tracing collection* uses a definition of 'garbage' that is decidable and may not include all objects that will never be accessed again. The second is that in practice, as we shall see in later chapters, garbage collector implementations

[4] "When C++ is your hammer, everything looks like a thumb," Steven M. Haflich, Chair of the NCITS/J13 technical committee for ANSI standard for Common Lisp.

[5] The final committee draft for the next ISO C++ standard is currently referred to as C++0x.

[6] `http://boost.org`

may choose for efficiency reasons not to reclaim some objects. Only the collector releases objects so the double-freeing problem cannot arise. All reclamation decisions are deferred to the collector, which has global knowledge of the structure of objects in the heap and the threads that can access them. The problems of *explicit deallocation* were largely due to the difficulty of making a global decision in a local context. Automatic dynamic memory management simply finesses this problem.

Above all, memory management is a software engineering issue. Well-designed programs are built from components (in the loosest sense of the term) that are highly cohesive and loosely coupled. Increasing the cohesion of modules makes programs easier to maintain. Ideally, a programmer should be able to understand the behaviour of a module from the code of that module alone, or at worst a few closely related modules. Reducing the coupling between modules means that the behaviour of one module is not dependent on the implementation of another module. As far as correct memory management is concerned, this means that modules should not have to know the rules of the memory management game played by other modules. In contrast, explicit memory management goes against sound software engineering principles of minimal communication between components; it clutters interfaces, either explicitly through additional parameters to communicate ownership rights, or implicitly by requiring programmers to conform to particular idioms. Requiring code to understand the rules of engagement limits the reusability of components.

The key argument in favour of garbage collection is not just that it simplifies coding — which it does — but that it uncouples the problem of memory management from interfaces, rather than scattering it throughout the code. It improves reusability. This is why garbage collection, in one form or another, has been a requirement of almost all modern languages (see Table 1.1). It is even expected that the next C++ standard will require code to be written so as to allow a garbage-collected implementation [Boehm and Spertus, 2009]. There is substantial evidence that managed code, including automatic memory management, reduces development costs [Butters, 2007]. Unfortunately, most of this evidence is anecdotal or compares development in different languages and systems (hence comparing more than just memory management strategies), and few detailed comparative studies have been performed. Nevertheless, one author has suggested that memory management should be the *prime* concern in the design of software for complex systems [Nagle, 1995]. Rovner [1985] estimated that 40% of development time for Xerox's Mesa system was spent on getting memory management correct. Possibly the strongest corroboration of the case for automatic dynamic memory management is an indirect, economic one: the continued existence of a wide variety of vendors and tools for detection of memory errors.

We do not claim that garbage collection is a silver bullet that will eradicate all memory-related programming errors or that it is applicable in all situations. Memory leaks are one of the most prevalent kinds of memory error. Although garbage collection tends to reduce the chance of memory leaks, it does not guarantee to eliminate them. If an object structure becomes unreachable to the rest of the program (for example, through any chain of pointers from the known roots), then the garbage collector will reclaim it. Since this is the only way that an object can be deleted, dangling pointers cannot arise. Furthermore, if deletion of an object causes its children to become unreachable, they too will be reclaimed. Thus, neither of the scenarios of Figure 1.1 are possible. However, garbage collection cannot guarantee the absence of space leaks. It has no answer to the problem of a data structure that *is* still reachable, but grows without limit (for example, if a programmer repeatedly adds data to a cache but never removes objects from that cache), or that is reachable and simply never accessed again.

Automatic dynamic memory management is designed to do just what it says. Some critics of garbage collection have complained that it is unable to provide general resource

ActionScript (2000)	Algol-68 (1965)	APL (1964)
AppleScript (1993)	AspectJ (2001)	Awk (1977)
Beta (1983)	C# (1999)	Cyclone (2006)
Managed C++ (2002)	Cecil (1992)	Cedar (1983)
Clean (1984)	CLU (1974)	D (2007)
Dart (2011)	Dylan (1992)	Dynace (1993)
E (1997)	Eiffel (1986)	Elasti-C (1997)
Emerald (1988)	Erlang (1990)	Euphoria (1993)
F# (2005)	Fortress (2006)	Green (1998)
Go (2010)	Groovy (2004)	Haskell (1990)
Hope (1978)	Icon (1977)	Java (1994)
JavaScript (1994)	Liana (1991)	Limbo (1996)
Lingo (1991)	Lisp (1958)	LotusScript (1995)
Lua (1994)	Mathematica (1987)	MATLAB (1970s)
Mercury (1993)	Miranda (1985)	ML (1990)
Modula-3 (1988)	Oberon (1985)	Objective-C (2007–)
Obliq (1993)	Perl (1986)	Pike (1996)
PHP (1995)	Pliant (1999)	POP-2 (1970)
PostScript (1982)	Prolog (1972)	Python (1991)
R (1993)	Rexx (1979)	Ruby (1993)
Sather (1990)	Scala (2003)	Scheme (1975)
Self (1986)	SETL (1969)	Simula (1964)
SISAL (1983)	Smalltalk (1972)	SNOBOL (1962)
Squeak (1996)	Tcl (1990)	Theta (1994)
VB.NET (2001)	VBScript (1996)	Visual Basic (1991)
VHDL (1987)	X10 (2004)	YAFL (1993)

Table 1.1: A selection of languages that rely on garbage collection.
Online sources: Dictionary of Programming Languages, Wikipedia and Google.

management, for example, to close files or windows promptly after their last use. However, this is unfair. Garbage collection is not a universal panacea. It attacks and solves a specific question: the management of memory resources. Nevertheless, the problem of general resource management in a garbage collected language is a substantial one. With explicitly-managed systems there is a straightforward and natural coupling between memory reclamation and the disposal of other resources. Automatic memory management introduces the problem of how to structure resource management in the absence of a natural coupling. However, it is interesting to observe that many resource release scenarios require something akin to a collector in order to detect whether the resource is still in use (reachable) from the rest of the program.

1.3 Comparing garbage collection algorithms

In this book we discuss a wide range of collectors, each designed with different workloads, hardware contexts and performance requirements in mind. Unfortunately, it is never possible to identify a 'best' collector for all configurations. For example, Fitzgerald and Tarditi [2000] found in a study of 20 benchmarks and six collectors that for every collector there was at least one benchmark that would run at least 15% faster with a more appropriate

collector. Singer *et al* [2007b] applied machine learning techniques to predict the best collector configuration for a particular program. Others have explored allowing *Java virtual machines* to switch collectors as they run if they believe that the characteristics of the workload being run would benefit from a different collector [Printezis, 2001; Soman *et al*, 2004]. In this section, we examine the metrics by which collectors can be compared. Nevertheless, such comparisons are difficult in both principle and practice. Details of implementation, *locality* and the practical significance of the constants in algorithmic complexity formulae make them less than perfect guides to practice. Moreover, the metrics are not independent variables. Not only does the performance of an algorithm depend on the topology and volume of objects in the heap, but also on the access patterns of the application. Worse, the tuning options in production virtual machines are inter-connected. Variation of one parameter to achieve a particular goal may lead to other, contradictory effects.

Safety

The prime consideration is that garbage collection should be *safe*: the collector must never reclaim the storage of live objects. However, safety comes with a cost, particularly for concurrent collectors (see Chapter 15). The safety of *conservative collection*, which receives no assistance from the compiler or run-time system, may in principle be vulnerable to certain compiler optimisations that disguise pointers [Jones, 1996, Chapter 9].

Throughput

A common goal for end users is that their programs should run faster. However, there are several aspects to this. One is that the overall time spent in garbage collection should be as low as possible. This is commonly referred to in the literature as the *mark/cons ratio*, comparing the early Lisp activities of the collector ('marking' live objects) and the mutator (creating or 'consing' new list cells). However, the user is most likely to want the application as a whole (mutator *plus* collector) to execute in as little time as possible. In most well designed configurations, much more CPU time is spent in the mutator than the collector. Therefore it may be worthwhile trading some collector performance for increased mutator throughput. For example, systems managed by mark-sweep collection occasionally perform more expensive compacting phases in order to reduce fragmentation so as to improve mutator allocation performance (and possibly mutator performance more generally).

Completeness and promptness

Ideally, garbage collection should be *complete*: eventually, all garbage in the heap should be reclaimed. However, this is not always possible nor even desirable. Pure reference counting collectors, for example, are unable to reclaim cyclic garbage (self-referential structures). For performance reasons, it may be desirable not to collect the whole heap at every *collection cycle*. For example, generational collectors segregate objects by their age into two or more regions called generations (we discuss generational garbage collection in Chapter 9). By concentrating effort on the youngest generation, generational collectors can both improve total collection time and reduce the average *pause time* for individual collections.

Concurrent collectors interleave the execution of mutators and collectors; the goal of such collectors is to avoid, or at least bound, interruptions to the user program. One consequence is that objects that become garbage after a collection cycle has started may not be reclaimed until the end of the next cycle; such objects are called *floating garbage*. Hence, in a

concurrent setting it may be more appropriate to define completeness as *eventual* reclamation of all garbage, as opposed to reclamation within one cycle. Different collection algorithms may vary in their *promptness* of reclamation, again leading to time/space trade-offs.

Pause time

On the other hand, an important requirement may be to minimise the collector's intrusion on program execution. Many collectors introduce pauses into a program's execution because they stop all mutator threads while collecting garbage. It is clearly desirable to make these pauses as short as possible. This might be particularly important for interactive applications or servers handling transactions (when failure to meet a deadline might lead to the transaction being retried, thus building up a backlog of work). However, mechanisms for limiting pause times may have side-effects, as we shall see in more detail in later chapters. For example, generational collectors address this goal by frequently and quickly collecting a small nursery region, and only occasionally collecting larger, older generations. Clearly, when tuning a generational collector, there is a balance to be struck between the sizes of the generations, and hence not only the pause times required to collect different generations but also the frequency of collections. However, because the sources of some inter-generational pointers must be recorded, generational collection imposes a small tax on pointer write operations by the mutator.

Parallel collectors stop the world to collect but reduce pause times by employing multiple threads. Concurrent and incremental collectors aim to reduce pause times still further by occasionally performing a small quantum of collection work interleaved or in parallel with mutator actions. This too requires taxation of the mutator in order to ensure correct synchronisation between mutators and collectors. As we shall see in Chapter 15, there are different ways to handle this synchronisation. The choice of mechanism affects both space and time costs. It also affects termination of a garbage collection cycle. The cost of the taxation on mutator time depends on how and which manipulations of the heap by the mutator (loads or stores) are recorded. The costs on space, and also collector termination, depends on how much floating garbage (see below) a system tolerates. Multiple mutator and collector threads add to the complexity. In any case, decreasing pause time will increase overall processing time (decrease processing rate).

Maximum or average pause times on their own are not adequate measures. It is also important that the mutator makes progress. The distribution of pause times is therefore also of interest. There are a number of ways that pause time distributions may be reported. The simplest might be a measure of variation such as standard deviation or a graphical representation of the distribution. More interesting measures include *minimum mutator utilisation* (MMU) and *bounded mutator utilisation* (BMU). Both the MMU [Cheng and Blelloch, 2001] and BMU [Sachindran *et al*, 2004] measures seek to display concisely the (minimum) fraction of time spent in the mutator, for any given time window. The x-axis of Figure 1.2 represents time, from 0 to total execution time, and its y-axis the fraction of CPU time spent in the mutator (utilisation). Thus, not only do MMU and BMU curves show total garbage collection time as a fraction of overall execution time (the y-intercept, at the top right of the curves is the mutators' overall share of processor time), but they also show the maximum pause time (the longest window for which the mutator's CPU utilisation is zero) as the x-intercept. In general, curves that are higher and more to the left are preferable since they tend towards a higher mutator utilisation for a smaller maximum pause. Note that the MMU is the *minimum* mutator utilisation (y) in any time window (x). As a consequence it is possible for a larger window to have a lower MMU than a smaller window, leading to dips in the curve. In contrast, BMU curves give the MMU in that time window or *any larger* one. Monotonically increasing BMU curves are perhaps more intuitive than MMU.

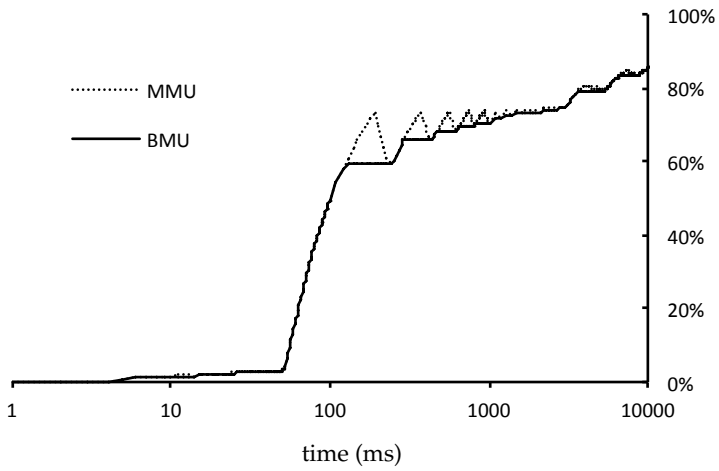


Figure 1.2: Minimum mutator utilisation and bounded mutator utilisation curves display concisely the (minimum) fraction of time spent in the mutator, for any given time window. MMU is the *minimum* mutator utilisation (y) in any time window (x) whereas BMU is minimum mutator utilisation in that time window or *any larger* one. In both cases, the x-intercept gives the maximum pause time and the y-intercept is the overall fraction of processor time used by the mutator.

Space overhead

The goal of memory management is safe and efficient use of space. Different memory managers, both explicit and automatic, impose different space overheads. Some garbage collectors may impose per-object space costs (for example, to store reference counts); others may be able to smuggle these overheads into objects' existing layouts (for example, a mark bit can often be hidden in a header word, or a forwarding pointer may be written over user data). Collectors may have a per-heap space overhead. For example, copying collectors divide the heap into two semispaces. Only one semispace is available to the mutator at any time; the other is held as a *copy reserve* into which the collector will evacuate live objects at collection time. Collectors may require auxiliary data structures. Tracing collectors need mark stacks to guide the traversal of the pointer graph in the heap; they may also store mark bits in separate bitmap tables rather than in the objects themselves. Concurrent collectors, or collectors that divide the heap into independently collected regions, require remembered sets that record where the mutator has changed the value of pointers, or the locations of pointers that span regions, respectively.

Optimisations for specific languages

Garbage collection algorithms can also be characterised by their applicability to different language paradigms. Functional languages in particular have offered a rich vein for optimisations related to memory management. Some languages, such as ML, distinguish mutable from immutable data. Pure functional languages, such as Haskell, go further and do not allow the user to modify any values (programs are *referentially transparent*). Internally, however, they typically update data structures at most once (from a 'thunk' to weak

head normal form); this gives multi-generation collectors opportunities to promote fully evaluated data structures eagerly (see Chapter 9). Authors have also suggested *complete* mechanisms for handling cyclic data structures with reference counting. Declarative languages may also allow other mechanisms for efficient management of heap spaces. Any data created in a logic language after a 'choice point' becomes unreachable after the program backtracks to that point. With a memory manager that keeps objects laid out in the heap in their order of allocation, memory allocated after the choice point can be reclaimed in constant time. Conversely, different language definitions may make specific requirements of the collector. The most notable are the ability to deal with a variety of pointer strengths and the need for the collector to cause dead objects to be finalised.

Scalability and portability

The final metrics we identify here are scalability and portability. With the increasing prevalence of multicore hardware on the desktop and even laptop (rather than just in large servers), it is becoming increasingly important that garbage collection can take advantage of the parallel hardware on offer. Furthermore, we expect parallel hardware to increase in scale (number of cores and sockets) and for heterogeneous processors to become more common. The demands on servers are also increasing, as heap sizes move into the tens or hundreds of gigabytes scale and as transaction loads increase. A number of collection algorithms depend on support from the operating system or hardware (for instance, by protecting pages or by double mapping virtual memory space, or on the availability of certain atomic operations on the processor). Such techniques are not necessarily portable.

1.4 A performance disadvantage?

We conclude the discussion of the comparative merits of automatic and manual dynamic memory management by asking if automatic memory management must be at a performance disadvantage compared with manual techniques. In general, the cost of automatic dynamic memory management is highly dependent on application behaviour and even hardware, making it impossible to offer simple estimates of overhead. Nevertheless, a long running criticism of garbage collection has been that it is slow compared to explicit memory management and imposes unacceptable overheads, both in terms of overall throughput and in pauses for garbage collection. While it is true that automatic memory management does impose a performance penalty on the program, it is not as much as is commonly assumed. Furthermore, explicit operations like `malloc` and `free` also impose a significant cost. Hertz, Feng, and Berger [2005] measured the true cost of garbage collection for a variety of Java benchmarks and collection algorithms. They instrumented a Java virtual machine to discover precisely when objects became unreachable, and then used the reachability trace as an oracle to drive a simulator, measuring cycles and cache misses. They compared a wide variety of garbage collector configurations against different implementations of `malloc/free`: the simulator invoked `free` at the point where the trace indicated that an object had become garbage. Although, as expected, results varied between both collectors and explicit allocators, Hertz *et al* found garbage collectors could match the execution time performance of explicit allocation provided they were given a sufficiently large heap (five times the minimum required). For more typical heap sizes, the garbage collection overhead increased to 17% on average.

1.5 Experimental methodology

One of the most welcome changes over the past decade or so has been the improvement in experimental methodology reported in the literature on memory management. Nevertheless, it remains clear that reporting standards in computer science have some way to improve before they match the quality of the very best practice in the natural or social sciences. Mytkowicz *et al* [2008] find measurement bias to be 'significant and commonplace'.

In a study of a large number of papers on garbage collection, Georges *et al* [2007] found the experimental methodology, even where reported, to be inadequately rigorous in many cases. Many reported performance improvements were sufficiently small, and the reports lacking in statistical analysis, to raise questions of whether any confidence could be placed in the results. Errors introduced may be systematic or random. Systematic errors are largely due to poor experimental practice and can often be reduced by more careful design of experiments. Random errors are typically due to non-determinism in the system under measurement. By their nature, these are unpredictable and often outside the experimenter's control; they should be addressed statistically.

The use of synthetic or small scale, 'toy', benchmarks has long been criticised as inadequate [Zorn, 1989]. Such benchmarks risk introducing systematic errors because they do not reflect the interactions in memory allocation that occur in real programs, or because their working sets are sufficiently small that they exhibit locality effects that real programs would not. Wilson *et al* [1995a] provide an excellent critique of such practices. Fortunately, other than for stress testing, synthetic and toy benchmarks have been largely abandoned in favour of larger scale benchmark suites, consisting of widely used programs that are believed to represent a wide range of typical behaviour (for example, the DaCapo suite for Java [Blackburn *et al*, 2006b]).

Experiments with benchmark suites that contain a large number of realistic programs can introduce systematic bias. *Managed run-times*, in particular, offer several opportunities for the introduction of systematic errors. Experimenters need to take care to distinguish the context that they are trying to examine: are they interested in start-up costs (important, for example, for short-lived programs) or in the steady state? For the latter, it is important to exclude system warm-up effects such as class loading and dynamic code optimisation. In both cases, it is probably important to disregard cold-start effects such as latency caused by loading the necessary files into the disk cache: thus Georges *et al* [2007] advocate running several invocations of the virtual machine and benchmark and discarding the first.

Dynamic (or run-time) compilation is a major source of non-determinism, and is particularly difficult to deal with when comparing alternative algorithms. One solution is to remove it. *Compiler replay* [Blackburn *et al*, 2006b] allows the user to record which methods are optimised and to which level in a preparatory run of the benchmark. This record can then used by the virtual machine to ensure the same level of optimisation in subsequent, performance runs. However, a problem with this approach is that alternative implementations typically execute different methods, particularly in the component under test. It is not clear which compilation record should be used. Two separate ones? Their intersection?

Sound experimental practice requires that outcomes are valid even in the presence of bias (for example, random errors). This requires repetitions of the experiment and statistical comparison of the results. To be able to state with confidence that one approach is superior to another requires that, first, a confidence level is stated, and second, confidence intervals for each alternative are derived from the results and that these intervals are not found to overlap. Georges *et al* [2007] offer a statistically rigorous methodology to address non-deterministic and unpredictable errors (including the effects of dynamic compilation). They advocate invoking one instance of the virtual machine and executing a benchmark

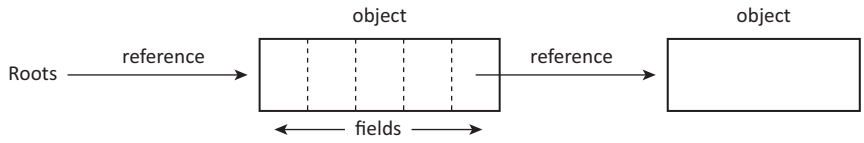


Figure 1.3: Roots, heap cells and references. Objects, denoted by rectangles, may be divided into a number of fields, delineated by dashed lines. References are shown as solid arrows.

many times until it reaches a steady state (that is, when the coefficient of variation[7] for the last k benchmark iterations falls below some preset threshold). These k iterations can then be used to compute a mean for the benchmark under steady state. By repeating this process, an overall mean and a confidence interval can be computed. Better, the whole distribution (or at least more than one or two moments of it) should be reported.

Garbage collection research needs thorough performance reports. A single 'spot' figure, even if decorated with a confidence interval, is not sufficient. The reason is that memory management involves space and time trade-offs. In most circumstances, one way to reduce collection times is to increase the size of the heap (up to a certain point — after that locality effects typically cause execution times to deteriorate). Thus, no experiment that reports a figure for just a single heap size can be taken seriously. It is vital, therefore, that environments allow the user to control the size of heaps (and spaces within those heaps) in order to understand fully the performance characteristics of a particular memory management algorithm. We firmly advocate this even for production virtual machines which may automatically adapt sizes for optimal performance; while automatic adaptation might be appropriate for end users, researchers and developers need more insight.

The chaotic nature of garbage collection reinforces this requirement. By calling garbage collection chaotic, we mean that small changes in configuration can, and commonly do, lead to large changes in behaviour. One example is the scheduling of collections. Even a small change to the point at which a garbage collection occurs may mean that a large data structure either remains reachable or becomes garbage. This can have large effects not only on the cost of the current collection but on how soon the next collection will occur, thus making such variation self-amplifying. By providing results for a range of heap sizes (often expressed in terms of multiples of the smallest heap size in which a program will run to completion), such 'jitter' is made readily apparent.

1.6 Terminology and notation

We conclude this chapter by explaining the notation used in the rest of the book. We also give more precise definitions of some of the terms used earlier.

First, a note about units of storage. We adopt the convention that a byte comprises eight bits. Similarly, we use *kilobyte* (KB), *megabyte* (MB), *gigabyte* (GB) and *terabyte* (TB) to mean a corresponding power of two multiple of the unit byte (2^{10}, 2^{20}, 2^{30}, 2^{40}, respectively), in flagrant disregard for the standard definitions of the SI decimal prefixes.

The heap

The heap is either a contiguous array of memory words or organised into a set of discontiguous blocks of contiguous words. A *granule* is the smallest unit of allocation, typically

[7] The coefficient of variation is the standard deviation divided by the mean.

a word or double-word, depending on *alignment* requirements. A *chunk* is a large contiguous group of granules. A *cell* is a generally smaller contiguous group of granules and may be allocated or free, or even wasted or unusable for some reason.

An *object* is a cell allocated for use by the application. An object is usually assumed to be a contiguous array of addressable bytes or words, divided into *slots* or *fields*, as in Figure 1.3 (although some memory managers for real-time or embedded systems may construct an individual large object as a pointer structure, this structure is not revealed to the user program). A field may contain a reference or some other *scalar* non-reference value such as an integer. A *reference* is either a pointer to a heap object or the distinguished value **`null`**. Usually, a reference will be the canonical *pointer* to the head of the object (that is, its first address), or it may point to some offset from the head. An object will sometimes also have a *header* field which stores metadata used by the run-time system, commonly (but not always) stored at the head of an object. A *derived pointer* is a pointer obtained by adding an offset to an object's canonical pointer. An *interior pointer* is a derived pointer to an internal object field.

A *block* is an aligned chunk of a particular size, usually a power of two. For completeness we mention also that a *frame* (when not referring to a *stack frame*) means a large 2^k sized portion of address space, and a space is a possibly discontiguous collection of chunks, or even objects, that receive similar treatment by the system. A *page* is as defined by the hardware and operating system's virtual memory mechanism, and a *cache line* (or *cache block*) is as defined by its *cache*. A *card* is a 2^k aligned chunk, smaller than a page, related to some schemes for remembering cross-space pointers (Section 11.8).

The heap is often characterised as an *object graph*, which is a *directed graph* whose *nodes* are heap objects and whose directed edges are the references to heap objects stored in their fields. An edge is a reference from a *source* node or a *root* (see below) to a *destination* node.

The mutator and the collector

Following Dijkstra *et al* [1976, 1978], a garbage-collected program is divided into two semi-independent parts.

- The *mutator* executes application code, which allocates new objects and mutates the object graph by changing reference fields so that they refer to different destination objects. These reference fields may be contained in heap objects as well as other places known as *roots*, such as static variables, thread stacks, and so on. As a result of such reference updates, any object can end up disconnected from the roots, that is, *unreachable* by following any sequence of edges from the roots.

- The *collector* executes garbage collection code, which discovers unreachable objects and reclaims their storage.

A program may have more than one mutator *thread*, but the threads together can usually be thought of as a single actor over the heap. Equally, there may be one or more collector threads.

The mutator roots

Separately from the heap memory, we assume some finite set of mutator roots, representing pointers held in storage that is *directly* accessible to the mutator without going through other objects. By extension, objects in the heap referred to directly by the roots are called *root objects*. The mutator visits objects in the graph by loading pointers from the current set of root objects (adding new roots as it goes). The mutator can also *discard* a root by

overwriting the root pointer's storage with some other reference (that is, **`null`** or a pointer to another object). We denote the set of (addresses of) the roots by `Roots`.

In practice, the roots usually comprise static/global storage and thread-local storage (such as thread stacks) containing pointers through which mutator threads can directly manipulate heap objects. As mutator threads execute over time, their state (and so their roots) will change.

In a type-safe programming language, once an object becomes unreachable in the heap, and the mutator has discarded all root pointers to that object, then there is no way for the mutator to reacquire a pointer to the object. The mutator cannot 'rediscover' the object arbitrarily (without interaction with the run-time system) — there is no pointer the mutator can traverse to it and arithmetic construction of new pointers is prohibited. A variety of languages support *finalisation* of at least some objects. These appear to the mutator to be 'resurrected' by the run-time system. Our point is that the mutator cannot gain access to any arbitrary unreachable object by its efforts alone.

References, fields and addresses

In general, we shall refer to a heap node N by using its memory address (though this need not necessarily be the initial word of an object, but may be to some appropriate standard point in the layout of the object's data and metadata). Given an object (at address) N, we can refer to arbitrary fields of the object — which may or may not contain pointers — by treating the object as an array of fields: the ith field of an object N will be denoted $N[i]$, counting fields from 0; the number of fields of N is written $|N|$. We write the usual C syntax for dereferencing a (non-null) pointer p as $*p$. Similarly, we use `&` to obtain the address of a field. Thus, we write $\&N[i]$ for the address of the ith field of N. Given an object (at address) N the set $\texttt{Pointers}(N)$ denotes the set of (addresses of) *pointer fields* of N. More formally:

$$\texttt{Pointers}(N) \;=\; \{a \mid a = \&N[i], \forall i : 0 \leq i < |N| \text{ where } N[i] \text{ is a pointer}\}$$

For convenience, we write `Pointers` to denote the set of all pointer fields of all objects in the heap. Similarly, `Nodes` denotes the set of all (allocated) objects in the heap. For convenience, we will also treat the set `Roots` as a pseudo-object (separate from the heap), and define $\texttt{Pointers}(\texttt{Roots}) = \texttt{Roots}$ synonymously. By implication, this allows us to write $\texttt{Roots}[i]$ to refer to the ith root field.

Liveness, correctness and reachability

An object is said to be *live* if it will be accessed at some time in the future execution of the mutator. A garbage collector is correct only if it never reclaims live objects. Unfortunately, *liveness* is an *undecidable* property of programs: there is no way to decide for an arbitrary program whether it will ever access a particular heap object or not.[8] Just because a program continues to hold a pointer to an object does not mean it will access it. Fortunately, we can approximate liveness by a property that is decidable: *pointer reachability*. An object N is *reachable* from an object M if N can be reached by following a chain of pointers, starting from some field f of M. By extension, an object is only usable by a mutator if there is a chain of pointers from one of the mutator's roots to the object.

More formally (in the mathematical sense that allows reasoning about reachability), we can define the immediate 'points-to' relation $\rightarrow_f$ as follows. For any two heap nodes M, N in `Nodes`, $M \rightarrow_f N$ if and only if there is some field location $f = \&M[i]$ in $\texttt{Pointers}(M)$

[8] The undecidability of liveness is a corollary of the halting problem.

such that $\star f{=}N$. Similarly, `Roots` $\rightarrow_f N$ if and only if there is some field f in `Roots` such that $\star f{=}N$. We say that N is *directly reachable* from M, written $M \rightarrow N$, if there is some field f in `Pointers`(M) such that $M \rightarrow_f N$ (that is, some field f of M points to N). Then, the set of reachable objects in the heap is the transitive referential closure from the set of `Roots` under the $\rightarrow$ relation, that is, the least set

$$\mathit{reachable} = \{N \in \texttt{Nodes} \mid (\exists r \in \texttt{Roots} : r \rightarrow N) \vee (\exists M \in \mathit{reachable} : M \rightarrow N)\} \quad (1.1)$$

An object that is unreachable in the heap, and not pointed to by any mutator root, can never be accessed by a type-safe mutator. Conversely, any object reachable from the roots may be accessed by the mutator. Thus, liveness is more profitably defined for garbage collectors by reachability. Unreachable objects are certainly dead and can safely be reclaimed. But any reachable object may still be live and must be retained. Although we realise that doing so is not strictly accurate, we will tend to use *live* and *dead* interchangeably with *reachable* and *unreachable*, and *garbage* as synonymous with unreachable.

Pseudo-code

We use a common pseudo-code to describe garbage collection algorithms. We offer these algorithm fragments as illustrative rather than definitive, preferring to resolve ambiguities informally in the text rather than formally in the pseudocode. Our goal is a concise and representative description of each algorithm rather than a full-fleshed implementation.

Indentation denotes the extent of procedure bodies and the scope of control statements. The assignment operator is $\leftarrow$ and the equality operator is $=$. Otherwise we use C-style symbols for the other logical and relational operators, such as `||` (conditional or), `&&` (conditional and), $\leq$, $\geq$, $\neq$, `%` (modulus) and so on.

The allocator

The heap *allocator*, which can be thought of as functionally orthogonal to the collector, supports two operations: `allocate`, which reserves the underlying memory storage for an object, and `free` which returns that storage to the allocator for subsequent re-use. The size of the storage reserved by `allocate` is passed as an optional parameter; when omitted the allocation is of a fixed-size object, or the size of the object is not necessary for understanding of the algorithm. Where necessary, we may pass further arguments to `allocate`, for example to distinguish arrays from other objects, or arrays of pointers from those that do not contain pointers, or to include other information necessary to initialise object headers.

Mutator read and write operations

As they execute, mutator threads perform several operations of interest to the collector: `New`, `Read` and `Write`. We adopt the convention of naming mutator operations with a leading upper-case letter, as opposed to lower-case for collector operations. Generally, these operations have the expected behaviour: allocating a new object, reading an object field or writing an object field. Specific memory managers may augment these basic operations with additional functionality that turns the operation into a *barrier*: an action that results in synchronous or asynchronous communication with the collector. We distinguish *read barriers* and *write barriers*.

New(). The New operation obtains a new heap object from the heap allocator which returns the address of the first word of the newly-allocated object. The mechanism for actual allocation may vary from one heap implementation to another, but collectors usually need to be informed that a given object has been allocated in order to initialise metadata for that object, and before it can be manipulated by the mutator. The trivial default definition of New simply allocates.

```
New():
    return allocate()
```

Read(**src**,**i**). The Read operation accesses an object field in memory (which may hold a scalar or a pointer) and returns the value stored at that location. Read generalises memory loads and takes two arguments: (a pointer to) the object and the (index of its) field being accessed. We allow src=Roots if the field src[i] is a root (that is, &src[i]∈Roots). The default, trivial definition of Read simply returns the contents of the field.

```
Read(src, i):
    return src[i]
```

Write(**src**,**i**,**val**). The Write operation modifies a particular location in memory. It generalises memory stores and takes three arguments: (a pointer to) the source object and the (index of its) field to be modified, plus the (scalar or pointer) value to be stored. Again, if src=Roots then the field src[i] is a root (that is, &src[i]∈Roots). The default, trivial definition of Write simply updates the field.

```
Write(src, i, val):
    src[i] ← val
```

Atomic operations

In the face of concurrency between mutator threads, collector threads, and between the mutator and collector, all collector algorithms require that certain code sequences appear to execute *atomically*. For example, stopping mutator threads makes the task of garbage collection appear to occur atomically: the mutator threads will never access the heap in the middle of garbage collection. Moreover, when running the collector concurrently with the mutator, the New, Read, and Write operations may need to appear to execute atomically with respect to the collector and/or other mutator threads. To simplify the exposition of collector algorithms we will usually leave implicit the precise mechanism by which atomicity of operations is achieved, simply marking them with the keyword **atomic**. The meaning is clear: all the steps of an atomic operation must appear to execute indivisibly and instantaneously with respect to other operations. That is, other operations will appear to execute either before or after the atomic operation, but never interleaved between any of the steps that constitute the atomic operation. For discussion of different techniques to achieve atomicity as desired see Chapter 11 and Chapter 13.

Sets, multisets, sequences and tuples

We use abstract data structures where this clarifies the discussion of an algorithm. We use mathematical notation where it is appropriate but does not obscure simpler concepts. For the most part, we will be interested in sets and tuples, and simple operations over them to add, remove or detect elements.

We use the usual definition of a *set* as a collection of distinct (that is, unique) elements. The *cardinality* of a set S, written $|S|$, is the number of its elements.

In addition to the standard set notation, we also make use of *multisets*. A multiset's elements may have repeated membership in the multiset. The cardinality of a multiset is the total number of its elements, including repeated memberships. The number of times an element appears is its *multiplicity*. We adopt the following notation:

- $[]$ denotes the empty multiset
- $[a, a, b]$ denotes the multiset containing two as and one b
- $[a, b] + [a] = [a, a, b]$ denotes multiset union
- $[a, a, b] - [a] = [a, b]$ denotes multiset subtraction

A *sequence* is an ordered list of elements. Unlike a set (or multiset), order matters. Like a multiset, the same element can appear multiple times at different positions in the sequence. We adopt the following notation:

- $()$ denotes the empty sequence
- (a, a, b) denotes the sequence containing two as followed by a b
- $(a, b) \cdot (a) = (a, b, a)$ denotes appending of the sequence (a) to (a, b)

While a *tuple* of length k can be thought of as being equivalent to a sequence of the same length, we sometimes find it convenient to use a different notation to emphasise the fixed length of a tuple as opposed to the variable length of a sequence, and so on. We adopt the notation below for tuples; we use tuples only of length two or more.

- $\langle a_1, ..., a_k \rangle$ denotes the k-tuple whose ith member is a_i, for $1 \leq i \leq k$

Bibliography

This bibliography contains over 400 references. However, our comprehensive database at http://www.cs.kent.ac.uk/~rej/gcbib/ contains over 2500 garbage collection related publications. This database can be searched online or downloaded as BIBTEX, PostScript or PDF. As well as details of the article, papers, books, theses and so on, the bibliography also contains abstracts for some entries and URLs or DOIs for most of the electronically available ones. We continually strive to keep this bibliography up to date as a service to the community. Here you can help: Richard (R.E.Jones@kent.ac.uk) would be very grateful to receive further entries (or corrections).

Santosh G. Abraham and Janak H. Patel. Parallel garbage collection on a virtual memory system. University Park, Pennsylvania, August 1987, pages 243–246. Pennsylvania State University Press. Also technical report CSRD 620, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development. 316, 317, 318, 326, 472

Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. In OOPSLA 2004, pages 224–236. doi: 10.1145/1028976.1028995. 32, 38, 46, 301, 302, 319

Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. WRL Research Report 95/7, Digital Western Research Laboratory, September 1995. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf. 237

Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996. doi: 10.1109/2.546611. 237

Ole Agesen. GC points in a threaded environment. Technical Report SMLI TR-98-70, Sun Microsystems Laboratories, Palo Alto, CA, 1998. http://dl.acm.org/citation.cfm?id=974974. 188, 189

Rafael Alonso and Andrew W. Appel. An advisor for flexible working sets. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990, pages 153–162. ACM Press. doi: 10.1145/98457.98753. 208, 209

Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987. doi: 10.1016/0020-0190(87)90175-X. 125, 171

Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989a. doi: 10.1002/spe.4380190206. 121, 122, 125, 195, 197

Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2: 153–162, 1989b. doi: 10.1007/BF01811537. 171, 172

Andrew W. Appel. Tutorial: Compilers and runtime systems for languages with garbage collection. In PLDI 1992. doi: 10.1145/143095. 113

Andrew W. Appel. Emulating write-allocate on a no-write-allocate cache. Technical Report TR-459-94, Department of Computer Science, Princeton University, June 1994. http://www.cs.princeton.edu/research/techreps/TR-459-94. 100, 165, 166

Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS–TR–450–94, Department of Computer Science, Princeton University, March 1994. http://www.cs.princeton.edu/research/techreps/TR-450-94. 171

Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, January 1996. doi: 10.1017/S095679680000157X. 171

Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In PLDI 1988, pages 11–20. doi: 10.1145/53990.53992. 316, 317, 318, 340, 352, 473

J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996. 146

Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In PLDI 2001, pages 168–179. doi: 10.1145/378795.378832. 412

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *10th ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, June 1998, pages 119–129. ACM Press. doi: 10.1145/277651.277678. 267, 268, 280

Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *8th ACM International Conference on Embedded Software*, Atlanta, GA, 2008, pages 245–254. ACM Press. doi: 10.1145/1450058.1450092. 400

Thomas H. Axford. Reference counting of cyclic graphs for functional programs. *Computer Journal*, 33(5):466–470, 1990. doi: 10.1093/comjnl/33.5.466. 67

Alain Azagury, Elliot K. Kolodner, and Erez Petrank. A note on the implementation of replication-based garbage collection for multithreaded applications and multiprocessor environments. *Parallel Processing Letters*, 9(3):391–399, 1999. doi: 10.1142/S0129626499000360. 342

Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *12th International Conference on Compiler Construction*, Warsaw, Poland, May 2003, pages 185–199. Volume 2622 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-36579-6_14. 369

Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In OOPSLA 2003, pages 269–281. doi: 10.1145/949305.949329. 331

Azul. Pauseless garbage collection. White paper AWP–005–020, Azul Systems Inc., July 2008. http://www.azulsystems.com/products/whitepaper/wp_pgc.pdf. 355, 361

Azul. Comparison of virtual memory manipulation metrics. White paper, Azul Systems Inc., 2010. http://www.managedruntime.org/files/downloads/AzulVmemMetricsMRI.pdf. 360

David F. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In Jørgen Lindskov Knudsen, editor, *15th European Conference on Object-Oriented Programming*, Budapest, Hungary, June 2001, pages 207–235. Volume 2072 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-45337-7_12. 59, 67, 72, 108, 157, 366, 373

David F. Bacon, Clement R. Attanasio, Han Bok Lee, V. T. Rajan, and Stephen E. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In PLDI 2001, pages 92–103. doi: 10.1145/378795.378819. 67, 366

David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In POPL 2003, pages 285–298. doi: 10.1145/604131.604155. 323, 391, 394, 395

David F. Bacon, Perry Cheng, and V.T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In LCTES 2003, pages 81–92. doi: 10.1145/780732.780744. 404

David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In OOPSLA 2004, pages 50–68. doi: 10.1145/1035292.1028982. 77, 80, 134

David F. Bacon, Perry Cheng, David Grove, and Martin T. Vechev. Syncopation: Generational real-time garbage collection in the Metronome. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, Chicago, IL, June 2005, pages 183–192. ACM SIGPLAN Notices 40(7), ACM Press. doi: 10.1145/1065910.1065937a. 399

Scott B. Baden. Low-overhead storage reclamation in the Smalltalk-80 virtual machine. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 331–342. Addison-Wesley, 1983. 61, 63

Brenda Baker, E. G. Coffman, and D. E. Willard. Algorithms for resolving conflicts in dynamic storage allocation. *Journal of the ACM*, 32(2):327–343, April 1985. doi: 10.1145/3149.335126. 139

Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978. doi: 10.1145/359460.359470. Also AI Laboratory Working Paper 139, 1977. 138, 277, 316, 317, 318, 337, 340, 341, 342, 347, 361, 377, 384, 385

Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992a. doi: 10.1145/130854.130862. 104, 139, 361

Henry G. Baker. CONS should not CONS its arguments, or a lazy alloc is a smart alloc. *ACM SIGPLAN Notices*, 27(3), March 1992b. doi: 10.1145/130854.130858. 147

Henry G. Baker. 'Infant mortality' and generational garbage collection. *ACM SIGPLAN Notices*, 28(4):55–57, April 1993. doi: 10.1145/152739.152747. 106

Jason Baker, Antonio Cunei, Filip Pizlo, and Jan Vitek. Accurate garbage collection in uncooperative environments with lazy pointer stacks. In *International Conference on Compiler Construction*, Braga, Portugal, March 2007. Volume 4420 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-71229-9_5. 436

Jason Baker, Antonio Cunei, Tomas Kalibera, Filip Pizlo, and Jan Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurrency and Computation: Practice and Experience*, 21(12):1572–1606, 2009. doi: 10.1002/cpe.1391. Supersedes Baker *et al* [2007]. 171

Katherine Barabash, Yoav Ossia, and Erez Petrank. Mostly concurrent garbage collection revisited. In OOPSLA 2003, pages 255–268. doi: 10.1145/949305.949328. 319, 320

Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, November 2005. doi: 10.1145/1108970.1108972. 284, 319, 320, 480

David A. Barrett and Benjamin Zorn. Garbage collection using a dynamic threatening boundary. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995, pages 301–314. ACM SIGPLAN Notices 30(6), ACM Press. doi: 10.1145/207110.207164. 123

David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In PLDI 1993, pages 187–196. doi: 10.1145/155090.155108. 114

Joel F. Bartlett. Compacting garbage collection with ambiguous roots. WRL Research Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988a. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-88-2.pdf. Also appears as Bartlett [1988b]. 30, 104

Joel F. Bartlett. Compacting garbage collection with ambiguous roots. *Lisp Pointers*, 1(6): 3–12, April 1988b. doi: 10.1145/1317224.1317225. 436

Joel F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical Note TN–12, DEC Western Research Laboratory, Palo Alto, CA, October 1989a. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-12.pdf. 170, 192

Joel F. Bartlett. SCHEME->C: a portable Scheme-to-C compiler. WRL Research Report 89/1, DEC Western Research Laboratory, Palo Alto, CA, January 1989b. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-89-1.pdf. 170

George Belotsky. C++ memory management: From fear to triumph. O'Reilly linuxdevcenter.com, July 2003. http://www.linuxdevcenter.com/pub/a/linux/2003/08/07/cpp_mm-3.html. 3

Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984. doi: 10.1145/579.587. 309

Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000, pages 117–128. ACM SIGPLAN Notices 35(11), ACM Press. doi: 10.1145/356989.357000. 102

Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, May 1977. doi: 1721.1/16428. Technical report MIT/LCS/TR–178. 103, 140

Stephen Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In PLDI 2008, pages 22–32. doi: 10.1145/1375581.1375586. 30, 100, 152, 153, 154, 159, 186

Stephen Blackburn, Robin Garner, Chris Hoffman, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiederman. The DaCapo benchmarks: Java benchmarking development and analysis (extended version). Technical report, The DaCapo Group, 2006a. http://dacapobench.org/dacapo-TR-CS-06-01.pdf. 59, 114, 125

Stephen M. Blackburn and Antony L. Hosking. Barriers: Friend or foe? In ISMM 2004, pages 143–151. doi: 10.1145/1029873.1029891. 202, 203

Stephen M. Blackburn and Kathryn S. McKinley. In or out? putting write barriers in their place. In ISMM 2002, pages 175–184. doi: 10.1145/512429.512452. 80

Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In OOPSLA 2003, pages 344–458. doi: 10.1145/949305.949336. 49, 55, 60, 61, 108, 157, 158, 159, 322

Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In OOPSLA 2001, pages 342–352. doi: 10.1145/504282.504307. 110, 132

Stephen M. Blackburn, Richard E. Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In PLDI 2002, pages 153–164. doi: 10.1145/512529.512548. 130, 131, 140, 202

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2004a, pages 25–36. ACM SIGMETRICS Performance Evaluation Review 32(1), ACM Press. doi: 10.1145/1005686.1005693. 46, 49, 54, 55, 79, 88, 105, 126, 130, 203

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *26th International Conference on Software Engineering*, Edinburgh, May 2004b, pages 137–146. IEEE Computer Society Press. doi: 10.1109/ICSE.2004.1317436. 26, 107, 116, 195, 196

Stephen M. Blackburn, Robin Garner, Chriss Hoffman, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, October 2006b, pages 169–190. ACM SIGPLAN Notices 41(10), ACM Press. doi: 10.1145/1167473.1167488. 10

Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. Profile-based pretenuring. *ACM Transactions on Programming Languages and Systems*, 29(1):1–57, 2007. doi: 10.1145/1180475.1180477. 110, 132

Bruno Blanchet. Escape analysis for object oriented languages: Application to Java. In OOPSLA 1999, pages 20–34. doi: 10.1145/320384.320387. 147

Ricki Blau. Paging on an object-oriented personal computer for Smalltalk. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Minneapolis, MN, August 1983, pages 44–54. ACM Press. doi: 10.1145/800040.801394. Also appears as Technical Report UCB/CSD 83/125, University of California, Berkeley, Computer Science Division (EECS). 50

Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In PLDI 1999, pages 104–117. doi: 10.1145/301618.301648. 256, 289, 292, 377, 378, 379, 380, 381, 382, 383, 384, 385, 391, 404, 406, 474, 483

Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980. doi: 10.1145/357103.357104. 67

Hans-Juergen Boehm. Mark-sweep vs. copying collection and asymptotic complexity, September 1995. http://www.hboehm.info/gc/complexity.html. 26

Hans-Juergen Boehm. Reducing garbage collector cache misses. In ISMM 2000, pages 59–64. doi: 10.1145/362422.362438. 23, 27

Hans-Juergen Boehm. Destructors, finalizers, and synchronization. In POPL 2003, pages 262–272. doi: 10.1145/604131.604153. 218, 219, 221

Hans-Juergen Boehm. The space cost of lazy reference counting. In *31st Annual ACM Symposium on Principles of Programming Languages*, Venice, Italy, January 2004, pages 210–219. ACM SIGPLAN Notices 39(1), ACM Press. doi: 10.1145/604131.604153. 59, 60

Hans-Juergen Boehm. Space efficient conservative garbage collection. In PLDI 1993, pages 197–206. doi: 10.1145/155090.155109. 105, 168

Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In PLDI 2008, pages 68–78. doi: 10.1145/1375581.1375591. 346

Hans-Juergen Boehm and Mike Spertus. Garbage collection in the next C++ standard. In ISMM 2009, pages 30–38. doi: 10.1145/1542431.1542437. 3, 4

Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988. doi: 10.1002/spe.4380180902. 22, 23, 31, 79, 94, 95, 96, 104, 137, 163, 166, 209, 280

Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In PLDI 1991, pages 157–164. doi: 10.1145/113445.113459. 202, 315, 316, 318, 323, 326, 472

Michael Bond and Kathryn McKinley. Tolerating memory leaks. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, October 2008, pages 109–126. ACM SIGPLAN Notices 43(10), ACM Press. doi: 10.1145/1449764.1449774. 208

Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In OOPSLA 2001, pages 353–366. doi: 10.1145/504282.504308. 209

Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. *ACM Transactions on Programming Languages and Systems*, 28(5):908–941, September 2006. doi: 10.1145/1152649.1152652. 209

R. P. Brent. Efficient implementation of the first-fit strategy for dynamic storage allocation. *ACM Transactions on Programming Languages and Systems*, 11(3):388–403, July 1989. doi: 10.1145/65979.65981. 139

Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In LFP 1984, pages 256–262. doi: 10.1145/800055.802042. 340, 341, 347, 361, 386, 404, 405

David R. Brownbridge. Cyclic reference counting for combinator machines. In Jean-Pierre Jouannaud, editor, *Conference on Functional Programming and Computer Architecture*, Nancy, France, September 1985, pages 273–288. Volume 201 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-15975-4_42. 67

F. Warren Burton. A buddy system variation for disk storage allocation. *Communications of the ACM*, 19(7):416–417, July 1976. doi: 10.1145/360248.360259. 96

Albin M. Butters. Total cost of ownership: A comparison of C/C++ and Java. Technical report, Evans Data Corporation, June 2007. http://costkiller.net/tribune/Tribu-PDF/Total-Cost-of-Ownership-A-Comparison-of-C-Cplusplus-and-Java.pdf. 1, 4

Brad Calder, Chandra Krintz, S. John, and T. Austin. Cache-conscious data placement. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998, pages 139–149. ACM SIGPLAN Notices 33(11), ACM Press. doi: 10.1145/291069.291036. 50

D. C. Cann and Rod R. Oldehoeft. Reference count and copy elimination for parallel applicative computing. Technical Report CS–88–129, Department of Computer Science, Colorado State University, Fort Collins, CO, 1988. 61

Dante Cannarozzi, Michael P. Plezbert, and Ron Cytron. Contaminated garbage collection. In PLDI 2000, pages 264–273. doi: 10.1145/349299.349334. 147

Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992. doi: 10.1145/142137.142141. 340

Patrick J. Caudill and Allen Wirfs-Brock. A third-generation Smalltalk-80 implementation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, November 1986, pages 119–130. ACM SIGPLAN Notices 21(11), ACM Press. doi: 10.1145/28697.28709. 114, 138

CC 2005. *14th International Conference on Compiler Construction*, Edinburgh, April 2005. Volume 3443 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/b107108. 456

Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin Hood hashing. In *26th Annual Symposium on Foundations of Computer Science*, Portland, OR, October 1985, pages 261–288. IEEE Computer Society Press. doi: 10.1109/SFCS.1985.48. 195

Yang Chang. *Garbage Collection for Flexible Hard Real-time Systems*. PhD thesis, University of York, 2007. 415

Yang Chang and Andy Wellings. Integrating hybrid garbage collection with dual priority scheduling. In *11th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2005, pages 185–188. IEEE Press, IEEE Computer Society Press. doi: 10.1109/RTCSA.2005.56. 415

Yang Chang and Andy Wellings. Low memory overhead real-time garbage collection for Java. In *4th International Workshop on Java Technologies for Real-time and Embedded Systems*, Paris, France, October 2006a. doi: 10.1145/1167999.1168014. 415

Yang Chang and Andy Wellings. Garbage collection for flexible hard real-time systems. *IEEE Transactions on Computers*, 59(8):1063–1075, August 2010. doi: 10.1109/TC.2010.13. 415

Yang Chang and Andy J. Wellings. Hard real-time hybrid garbage collection with low memory requirements. In *27th IEEE Real-Time Systems Symposium*, December 2006b, pages 77–86. doi: 10.1109/RTSS.2006.25. 415

David R. Chase. *Garbage Collection and Other Optimizations*. PhD thesis, Rice University, August 1987. doi: 1911/16127. 104

David R. Chase. Safety considerations for storage allocation optimizations. In PLDI 1988, pages 1–10. doi: 10.1145/53990.53991. 104

A. M. Cheadle, A. J. Field, and J. Nyström-Persson. A method specialisation and virtualised execution environment for Java. In David Gregg, Vikram Adve, and Brian Bershad, editors, *4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Seattle, WA, March 2008, pages 51–60. ACM Press. doi: 10.1145/1346256.1346264. 170, 341

Andrew M. Cheadle, Anthony J. Field, Simon Marlow, Simon L. Peyton Jones, and R.L While. Non-stop Haskell. In *5th ACM SIGPLAN International Conference on Functional Programming*, Montreal, September 2000, pages 257–267. ACM Press. doi: 10.1145/351240.351265. 170, 171

Andrew M. Cheadle, Anthony J. Field, Simon Marlow, Simon L. Peyton Jones, and Lyndon While. Exploring the barrier to entry — incremental generational garbage collection for Haskell. In ISMM 2004, pages 163–174. doi: 10.1145/1029873.1029893. 99, 170, 171, 340, 341

Wen-Ke Chen, Sanjay Bhansali, Trishul M. Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization. In PLDI 2006, pages 332–340. doi: 10.1145/1133981.1134021. 53

C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13 (11):677–8, November 1970. doi: 10.1145/362790.362798. 43, 44

Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In PLDI 2001, pages 125–136. doi: 10.1145/378795.378823. 7, 187, 289, 290, 304, 377, 382, 384, 474

Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998, pages 162–173. ACM SIGPLAN Notices 33(5), ACM Press. doi: 10.1145/277650.277718. 110, 132, 146

Perry Sze-Din Cheng. *Scalable Real-Time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, Carnegie Mellon University, September 2001. http://reports-archive.adm.cs.cmu.edu/anon/2001/CMU-CS-01-174.pdf. SCS Technical Report CMU-CS-01-174. 289, 382, 474

Chen-Yong Cher, Antony L. Hosking, and T.N. Vijaykumar. Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. In Shubu Mukherjee and Kathryn S. McKinley, editors, *11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 2004, pages 199–210. ACM SIGPLAN Notices 39(11), ACM Press. doi: 10.1145/1024393.1024417. 27, 51

Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In ISMM 1998, pages 37–48. doi: 10.1145/301589.286865. 53

Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In PLDI 1999, pages 1–12. doi: 10.1145/301618.301633. 50

Hyeonjoong Cho, Chewoo Na, Binoy Ravindran, and E. Douglas Jensen. On scheduling garbage collector in dynamic real-time systems with statistical timing assurances. *Real-Time Systems*, 36(1–2):23–46, 2007. doi: 10.1007/s11241-006-9011-0. 415

Hyeonjoong Cho, Binoy Ravindran, and Chewoo Na. Garbage collector scheduling in dynamic, multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):845–856, June 2009. doi: 10.1109/TPDS.2009.20. 415

Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Riviere. Evaluation of AMD's Advanced Synchronization Facility within a complete transactional memory stack. In *European Conference on Computer Systems (EuroSys)*, 2010, pages 27–40. ACM Press. doi: 10.1145/1755913.1755918. 271

Douglas W. Clark and C. Cordell Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–86, February 1977. doi: 10.1145/359423.359427. 73

Cliff Click, Gil Tene, and Michael Wolf. The Pauseless GC algorithm. In VEE 2005, pages 46–56. doi: 10.1145/1064979.1064988. 355, 361

Marshall P. Cline and Greg A. Lomow. *C++ FAQs: Frequently Asked Questions*. Addison-Wesley, 1995. 3

William D. Clinger and Lars T. Hansen. Generational garbage collection and the radioactive decay model. In PLDI 1997, pages 97–108. doi: 10.1145/258915.258925. 128

Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4): 532–553, 1983. doi: 10.1145/69575.357226. 34

George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960. doi: 10.1145/367487.367501. xxiii, 2, 57

W. T. Comfort. Multiword list items. *Communications of the ACM*, 7(6):357–362, June 1964. doi: 10.1145/512274.512288. 94

Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In LFP 1992, pages 43–52. doi: 10.1145/141471.141501. 209

Erik Corry. Optimistic stack allocation for Java-like languages. In ISMM 2006, pages 162–173. doi: 10.1145/1133956.1133978. 147

Jim Crammond. A garbage collection algorithm for shared memory parallel processors. *International Journal Of Parallel Programming*, 17(6):497–522, 1988. doi: 10.1007/BF01407816. 299, 300

David Detlefs. Automatic inference of reference-count invariants. In *2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, Venice, Italy, January 2004a. http://www.diku.dk/topps/space2004. 152

David Detlefs. A hard look at hard real-time garbage collection. In *7th International Symposium on Object-Oriented Real-Time Distributed Computing*, Vienna, May 2004b, pages 23–32. IEEE Press. doi: 10.1109/ISORC.2004.1300325. Invited paper. 377, 385

David Detlefs, William D. Clinger, Matthias Jacob, and Ross Knippel. Concurrent remembered set refinement in generational garbage collection. In *2nd Java Virtual Machine Research and Technology Symposium*, San Francisco, CA, August 2002a. USENIX. https://www.usenix.org/conference/java-vm-02/concurrent-remembered-set-refinement-generational-garbage-collection. 196, 197, 199, 201, 319

David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. Garbage-first garbage collection. In ISMM 2004, pages 37–48. doi: 10.1145/1029873.1029879. 150, 159

David L. Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie Mellon University, Pittsburgh, PA, May 1990. http://repository.cmu.edu/compsci/1956. 340

David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele. Lock-free reference counting. In *20th ACM Symposium on Distributed Computing*, Newport, Rhode Island, August 2001, pages 190–199. ACM Press. doi: 10.1145/383962.384016. 365

David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele. Lock-free reference counting. *Distributed Computing*, 15:255–271, 2002b. doi: 10.1007/s00446-002-0079-z. 365

John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-64.pdf. 338, 340, 366

L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976. doi: 10.1145/360336.360345. 61, 105

Sylvia Dieckmann and Urs Hölzle. The allocation behaviour of the SPECjvm98 Java benchmarks. In Rudolf Eigenman, editor, *Performance Evaluation and Benchmarking with Realistic Applications*, chapter 3, pages 77–108. MIT Press, 2001. 59

Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In Rachid Guerraoui, editor, *13th European Conference on Object-Oriented Programming*, Lisbon, Portugal, July 1999, pages 92–115. Volume 1628 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-48743-3_5. 59, 114, 125

Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Language Hierarchies and Interfaces: International Summer School*, volume 46 of *Lecture Notes in Computer Science*, pages 43–56. Springer-Verlag, Marktoberdorf, Germany, 1976. doi: 10.1007/3-540-07994-7_48. 12, 20, 309, 315, 316, 323, 472, 473

Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978. doi: 10.1145/359642.359655. 12, 20, 309, 315, 316, 317, 318, 323, 330

Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000. doi: 10.1147/sj.391.0151. 30, 100, 101, 150, 151, 152, 186

Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In PLDI 1992, pages 273–282. doi: 10.1145/143095.143140. 179, 180, 183, 184

Amer Diwan, David Tarditi, and J. Eliot B. Moss. Memory subsystem performance of programs using copying garbage collection. In POPL 1994, pages 1–14. doi: 10.1145/174675.174710. 100, 165

Julian Dolby. Automatic inline allocation of objects. In PLDI 1997, pages 7–17. doi: 10.1145/258915.258918. 148

Julian Dolby and Andrew A. Chien. An automatic object inlining optimization and its evaluation. In PLDI 2000, pages 345–357. doi: 10.1145/349299.349344. 148

Julian Dolby and Andrew A. Chien. An evaluation of automatic object inline allocation techniques. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 1998, pages 1–20. ACM SIGPLAN Notices 33(10), ACM Press. doi: 10.1145/286936.286943. 148

Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In POPL 1994, pages 70–83. doi: 10.1145/174675.174673. 107, 108, 329, 331, 369

Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *20th Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993, pages 113–123. ACM Press. doi: 10.1145/158511.158611. 107, 108, 146, 329

Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In PLDI 2000, pages 274–284. doi: 10.1145/349299.349336. 330, 331

Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In ISMM 2002, pages 76–87. doi: 10.1145/512429.512439. 109, 110, 146

Kevin Donnelly, Joe Hallett, and Assaf Kfoury. Formal semantics of weak references. In ISMM 2006, pages 126–137. doi: 10.1145/1133956.1133974. 228

R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector. In PLDI 1993, pages 207–216. doi: 10.1145/155090.155110. 220

ECOOP 2007, Erik Ernst, editor. *21st European Conference on Object-Oriented Programming*, Berlin, Germany, July 2007. Volume 4609 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-73589-2. 445, 466

Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. In *USENIX C++ Conference*, Portland, OR, August 1992. USENIX. 59, 74

Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *ACM/IEEE Conference on Supercomputing*, San Jose, CA, November 1997. doi: 10.1109/SC.1997.10059. 249, 277, 280, 281, 283, 289, 299, 304, 480

A. P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8): 3–6, August 1958. doi: 10.1145/368892.368907. 169

Shahrooz Feizabadi and Godmar Back. Java garbage collection scheduling in utility accrual scheduling environments. In *3rd International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, San Diego, CA, 2005. http://people.cs.vt.edu/~gback/papers/jtres2005-cadus.pdf. 415

Shahrooz Feizabadi and Godmar Back. Garbage collection-aware sheduling utility accrual scheduling environments. *Real-Time Systems*, 36(1–2), July 2007. doi: 10.1007/s11241-007-9020-7. 415

Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969. doi: 10.1145/363269.363280. 43, 44, 50, 107

Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *1st International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, March 2003, pages 241–252. IEEE Computer Society Press. doi: 10.1109/CGO.2003.1191549. 190

David A. Fisher. Bounded workspace garbage collection in an address order preserving list processing environment. *Information Processing Letters*, 3(1):29–32, July 1974. doi: 10.1016/0020-0190(74)90044-1. 36

Robert Fitzgerald and David Tarditi. The case for profile-directed selection of garbage collectors. In ISMM 2000, pages 111–120. doi: 10.1145/362422.362472. 5, 77, 138, 202

Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In JVM 2001. http://www.usenix.org/events/jvm01/flood.html. 34, 36, 248, 278, 280, 282, 283, 284, 288, 289, 292, 298, 300, 301, 303, 304, 357, 471, 474, 480

John K. Foderaro and Richard J. Fateman. Characterization of VAX Macsyma. In *1981 ACM Symposium on Symbolic and Algebraic Computation*, Berkeley, CA, 1981, pages 14–19. ACM Press. doi: 10.1145/800206.806364. 113

John K. Foderaro, Keith Sklower, Kevin Layer, *et al*. *Franz Lisp Reference Manual*. Franz Inc., 1985. 27

Daniel Frampton, David F. Bacon, Perry Cheng, and David Grove. Generational real-time garbage collection: A three-part invention for young objects. In ECOOP 2007, pages 101–125. doi: 10.1007/978-3-540-73589-2_6. 399

Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960. doi: 10.1145/367390.367400. 413

Daniel P. Friedman and David S. Wise. Reference counting can manage the circular environments of mutual recursion. *Information Processing Letters*, 8(1):41–45, January 1979. doi: 10.1016/0020-0190(79)90091-7. 66, 67

Robin Garner, Stephen M. Blackburn, and Daniel Frampton. Effective prefetch for mark-sweep garbage collection. In ISMM 2007, pages 43–54. doi: 10.1145/1296907.1296915. 23, 26, 28, 29

Alex Garthwaite. *Making the Trains Run On Time*. PhD thesis, University of Pennsylvania, 2005. 194

Alex Garthwaite, Dave Dice, and Derek White. Supporting per-processor local-allocation buffers using lightweight user-level preemption notification. In VEE 2005, pages 24–34. doi: 10.1145/1064979.1064985. 101, 195

Alexander T. Garthwaite, David L. Detlefs, Antonios Printezis, and Y. Srinivas Ramakrishna. Method and mechanism for finding references in a card in time linear in the size of the card in a garbage-collected heap. United States Patent 7,136,887 B2, Sun Microsystems, November 2006. 200, 201

David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *9th International Conference on Compiler Construction*, Berlin, April 2000, pages 82–93. Volume 2027 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-46423-9_6. 147, 148

GC 1990, Eric Jul and Niels-Christian Juul, editors. *OOPSLA/ECOOP Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, Canada, October 1990. ftp://ftp.cs.utexas.edu/pub/garbage/GC90/. 449, 465

GC 1991, Paul R. Wilson and Barry Hayes, editors. *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. `ftp://ftp.cs.utexas.edu/pub/garbage/GC91/`. 449, 466

GC 1993, J. Eliot B. Moss, Paul R. Wilson, and Benjamin Zorn, editors. *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems*, October 1993. `ftp://ftp.cs.utexas.edu/pub/garbage/GC93/`. 448, 449, 465

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Montréal, Canada, October 2007, pages 57–76. ACM SIGPLAN Notices 42(10), ACM Press. doi: `10.1145/1297027.1297033`. 10

Joseph (Yossi) Gil and Itay Maman. Micro patterns in Java code. In OOPSLA 2005, pages 97–116. doi: `10.1145/1094811.1094819`. 132

O. Goh, Yann-Hang Lee, Z. Kaakani, and E. Rachlin. Integrated scheduling with garbage collection for real-time embedded applications in CLI. In *9th International Symposium on Object-Oriented Real-Time Distributed Computing*, Gyeongju, Korea, April 2006. IEEE Press. doi: `10.1109/ISORC.2006.41`. 415

Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In PLDI 1991 [PLDI 1991], pages 165–176. doi: `10.1145/113445.113460`. 171, 172

Benjamin Goldberg. Incremental garbage collection without tags. In *European Symposium on Programming*, Rennes, France, February 1992, pages 200–218. Volume 582 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: `10.1007/3-540-55253-7_12`. 171

Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In LFP 1992, pages 53–65. doi: `10.1145/141471.141504`. 171, 172

Marcelo J. R. Gonçalves and Andrew W. Appel. Cache performance of fast-allocating programs. In *Conference on Functional Programming and Computer Architecture*, La Jolla, CA, June 1995, pages 293–305. ACM Press. doi: `10.1145/224164.224219`. 165

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Addison Wesley, Java SE 8 edition, February 2015. `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`. 346

Eiichi Goto. Monocopy and associative algorithms in an extended LISP. Technical Report 74-03, Information Science Laboratories, Faculty of Science, University of Tokyo, 1974. 169

David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977. doi: `10.1145/359897.359903`. 462

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In PLDI 2002, pages 282–293. doi: `10.1145/512529.512563`. 106

Chris Grzegorczyk, Sunil Soman, Chandra Krintz, and Rich Wolski. Isla Vista heap sizing: Using feedback to avoid paging. In *5th International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, March 2007, pages 325–340. IEEE Computer Society Press. doi: `10.1109/CGO.2007.20`. 209

Samuel Guyer and Kathryn McKinley. Finding your cronies: Static analysis for dynamic object colocation. In OOPSLA 2004, pages 237–250. doi: 10.1145/1028976.1028996. 110, 132, 143

Robert H. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In LFP 1984, pages 9–17. doi: 10.1145/800055.802017. 277, 294

Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985. doi: 10.1145/4472.4478. 277, 338, 342, 345

Lars Thomas Hansen. *Older-first Garbage Collection in Practice*. PhD thesis, Northeastern University, November 2000. 128

Lars Thomas Hansen and William D. Clinger. An experimental study of renewal-older-first garbage collection. In *7th ACM SIGPLAN International Conference on Functional Programming*, Pittsburgh, PA, September 2002, pages 247–258. ACM SIGPLAN Notices 37(9), ACM Press. doi: 10.1145/581478.581502. 128

David R. Hanson. Storage management for an implementation of SNOBOL4. *Software: Practice and Experience*, 7(2):179–192, 1977. doi: 10.1002/spe.4380070206. 41

Tim Harris and Keir Fraser. Language support for lightweight transactions. In OOPSLA 2003, pages 388–402. doi: 10.1145/949305.949340. 272

Timothy Harris. Dynamic adaptive pre-tenuring. In ISMM 2000, pages 127–136. doi: 10.1145/362422.362476. 132

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In Dahlia Malkhi, editor, *International Conference on Distributed Computing*, Toulouse, France, October 2002, pages 265–279. Volume 2508 of *Lecture Notes in Computer Science*. doi: 10.1007/3-540-36108-1_18. 406

Pieter H. Hartel. *Performance Analysis of Storage Management in Combinator Graph Reduction*. PhD thesis, Department of Computer Systems, University of Amsterdam, Amsterdam, 1988. 73

Barry Hayes. Using key object opportunism to collect old objects. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Phoenix, AZ, November 1991, pages 33–46. ACM SIGPLAN Notices 26(11), ACM Press. doi: 10.1145/117954.117957. 23, 100, 114

Barry Hayes. Finalization in the collector interface. In IWMM 1992, pages 277–298. doi: 10.1007/BFb0017196. 221

Barry Hayes. Ephemerons: A new finalization mechanism. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Atlanta, GA, October 1997, pages 176–183. ACM SIGPLAN Notices 32(10), ACM Press. doi: 10.1145/263698.263733. 227

Laurence Hellyer, Richard Jones, and Antony L. Hosking. The locality of concurrent write barriers. In ISMM 2010, pages 83–92. doi: 10.1145/1806651.1806666. 316

Fergus Henderson. Accurate garbage collection in an uncooperative environment. In ISMM 2002, pages 150–156. doi: 10.1145/512429.512449. 171

Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998. http://www.dna.lth.se/home/Roger_Henriksson/. 377, 386, 387, 388, 389, 390, 391, 393, 399

Maurice Herlihy and J. Eliot B Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992. doi: 10.1109/71.139204. 249, 342, 343, 344, 345, 361

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufman, April 2008. xxiii, 2, 229, 240, 243, 254, 255, 256

Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3): 463–492, 1990. doi: 10.1145/78969.78972. 254

Maurice P. Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Annual International Symposium on Computer Architecture*, San Diego, CA, May 1993, pages 289–300. IEEE Press. doi: 10.1145/165123.165164. 270, 344

Maurice P. Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures. In *16th International Symposium on Distributed Computing*, Toulouse, France, October 2002, pages 339–353. Volume 2508 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-36108-1_23. 374

Matthew Hertz. *Quantifying and Improving the Performance of Garbage Collection*. PhD thesis, University of Massachusetts, September 2006. http://www-cs.canisius.edu/~hertzm/thesis.pdf. 208

Matthew Hertz and Emery Berger. Quantifying the performance of garbage collection vs. explicit memory management. In OOPSLA 2005, pages 313–326. doi: 10.1145/1094811.1094836. 30, 55, 79

Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In Vivek Sarkar and Mary W. Hall, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005, pages 143–153. ACM SIGPLAN Notices 40(6), ACM Press. doi: 10.1145/1064978.1065028. 9, 108, 110, 156, 208

Matthew Hertz, Jonathan Bard, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Kirk Kelsey, and Chen Ding. Waste not, want not — resource-based garbage collection in a shared environment. Technical Report TR–951, The University of Rochester, December 2009. doi: 1802/8838. 210

D. S. Hirschberg. A class of dynamic memory allocation algorithms. *Communications of the ACM*, 16(10):615–618, October 1973. doi: 10.1145/362375.362392. 96

Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In OOPSLA 2003, pages 359–373. doi: 10.1145/949305.949337. 143, 144, 158

Urs Hölzle. A fast write barrier for generational garbage collectors. In GC 1993. ftp://ftp.cs.utexas.edu/pub/garbage/GC93/hoelzle.ps. 197, 198

Antony L Hosking. Portable, mostly-concurrent, mostly-copying garbage collection for multi-processors. In ISMM 2006, pages 40–51. doi: 10.1145/1133956.1133963. 30, 340

Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In GC 1993. ftp://ftp.cs.utexas.edu/pub/garbage/GC93/hosking.ps. 201

Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *14th ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993, pages 106–119. ACM SIGOPS Operating Systems Review 27(5), ACM Press. doi: 10.1145/168619.168628. 353

Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 1992, pages 92–109. ACM SIGPLAN Notices 27(10), ACM Press. doi: 10.1145/141936.141946. 137, 138, 193, 194, 195, 197, 199, 201, 202, 206

Antony L. Hosking, Nathaniel Nystrom, Quintin Cutts, and Kumar Brahnmath. Optimizing the read and write barrier for orthogonal persistence. In Ronald Morrison, Mick J. Jordan, and Malcolm P. Atkinson, editors, *8th International Workshop on Persistent Object Systems (August, 1998)*, Tiburon, CA, 1999, pages 149–159. Advances in Persistent Object Systems, Morgan Kaufmann. http://hosking.github.io/links/Hosking+1998POS.pdf. 323

Xianlong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Z. Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In OOPSLA 2004, pages 69–80. doi: 10.1145/1028976.1028983. 52, 170

Richard L. Hudson. Finalization in a garbage collected world. In GC 1991. ftp://ftp.cs.utexas.edu/pub/garbage/GC91/hudson.ps. 221

Richard L. Hudson and Amer Diwan. Adaptive garbage collection for Modula-3 and Smalltalk. In GC 1990. ftp://ftp.cs.utexas.edu/pub/garbage/GC90/Hudson.ps.Z. 195

Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM-ISCOPE Conference on Java Grande*, Palo Alto, CA, June 2001, pages 48–57. ACM Press. doi: 10.1145/376656.376810. 346, 361

Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3–5): 223–261, 2003. doi: 10.1002/cpe.712. 346, 351, 361

Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In IWMM 1992, pages 388–403. doi: 10.1007/BFb0017203. 109, 130, 137, 140, 143, 158, 202, 208

Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. Technical Report COINS 91-47, University of Massachusetts, September 1991. https://web.cs.umass.edu/publication/docs/1991/UM-CS-1991-047.pdf. 118, 138

R. John M. Hughes. A semi-incremental garbage collection algorithm. *Software: Practice and Experience*, 12(11):1081–1082, November 1982. doi: 10.1002/spe.4380121108. 25, 26

Akira Imai and Evan Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *Transactions on Parallel and Distributed Systems*, 4(9): 1030–1040, 1993. doi: 10.1109/71.243529. 294, 295, 296, 297, 304, 474

ISMM 1998, Simon L. Peyton Jones and Richard Jones, editors. *1st International Symposium on Memory Management*, Vancouver, Canada, October 1998. ACM SIGPLAN Notices 34(3), ACM Press. doi: 10.1145/286860. 441, 457, 460

ISMM 2000, Craig Chambers and Antony L. Hosking, editors. *2nd International Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM SIGPLAN Notices 36(1), ACM Press. doi: 10.1145/362422. 438, 445, 447, 458, 462, 463

ISMM 2002, Hans-J. Boehm and David Detlefs, editors. *3rd International Symposium on Memory Management*, Berlin, Germany, June 2002. ACM SIGPLAN Notices 38(2 supplement), ACM Press. doi: 10.1145/773146. 437, 444, 447, 451, 458

ISMM 2004, David F. Bacon and Amer Diwan, editors. *4th International Symposium on Memory Management*, Vancouver, Canada, October 2004. ACM Press. doi: 10.1145/1029873. 437, 440, 442, 451, 456, 459, 461, 466

ISMM 2006, Erez Petrank and J. Eliot B. Moss, editors. *5th International Symposium on Memory Management*, Ottawa, Canada, June 2006. ACM Press. doi: 10.1145/1133956. 442, 444, 449, 454, 460, 461, 466

ISMM 2007, Greg Morrisett and Mooly Sagiv, editors. *6th International Symposium on Memory Management*, Montréal, Canada, October 2007. ACM Press. doi: 10.1145/1296907. 445, 453, 457, 460

ISMM 2008, Richard Jones and Steve Blackburn, editors. *7th International Symposium on Memory Management*, Tucson, AZ, June 2008. ACM Press. doi: 10.1145/1375634. 451, 453, 460

ISMM 2009, Hillel Kolodner and Guy Steele, editors. *8th International Symposium on Memory Management*, Dublin, Ireland, June 2009. ACM Press. doi: 10.1145/1542431. 438, 455, 464

ISMM 2010, Jan Vitek and Doug Lea, editors. *9th International Symposium on Memory Management*, Toronto, Canada, June 2010. ACM Press. doi: 10.1145/1806651. 447, 460

ISMM 2011, Hans Boehm and David Bacon, editors. *10th International Symposium on Memory Management*, San Jose, CA, June 2011. ACM Press. doi: 10.1145/1993478. 463

IWMM 1992, Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, St Malo, France, 17–19 September 1992. Volume 637 of *Lecture Notes in Computer Science*, Springer. doi: 10.1007/BFb0017181. 447, 449, 452, 455

IWMM 1995, Henry G. Baker, editor. *International Workshop on Memory Management*, Kinross, Scotland, 27–29 September 1995. Volume 986 of *Lecture Notes in Computer Science*, Springer. doi: 10.1007/3-540-60368-9. 453, 466

Erik Johansson, Konstantinos Sagonas, and Jesper Wilhelmsson. Heap architectures for concurrent languages using message passing. In ISMM 2002, pages 88–99. doi: 10.1145/512429.512440. 146

Mark S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, December 1997. ftp://ftp.cs.utexas.edu/pub/garbage/johnstone-dissertation.ps.gz. 152

Richard Jones and Chris Ryder. A study of Java object demographics. In ISMM 2008, pages 121–130. doi: 10.1145/1375634.1375652. 23, 106, 113, 114

Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. http://www.cs.kent.ac.uk/people/staff/rej/gcbook/gcbook.html. With a chapter on Distributed Garbage Collection by R. Lins. xxiv, xxv, 6, 17, 30, 42, 54, 67, 79, 117, 119, 124, 138, 142, 150, 167

Richard E. Jones and Andy C. King. Collecting the garbage without blocking the traffic. Technical Report 18–04, Computing Laboratory, University of Kent, September 2004. http://www.cs.kent.ac.uk/pubs/2004/1970/. This report summarises King [2004]. 451

Richard E. Jones and Andy C. King. A fast analysis for thread-local garbage collection with dynamic class loading. In *5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, Budapest, September 2005, pages 129–138. IEEE Computer Society Press. doi: 10.1109/SCAM.2005.1. This is a shorter version of Jones and King [2004]. 107, 109, 145, 146, 159

H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1): 26–30, July 1979. doi: 10.1016/0020-0190(79)90103-0. 32, 37, 38

Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. Dynamic object sampling for pretenuring. In ISMM 2004, pages 152–162. doi: 10.1145/1029873.1029892. 132

JVM 2001. *1st Java Virtual Machine Research and Technology Symposium*, Monterey, CA, April 2001. USENIX. https://www.usenix.org/legacy/event/jvm01/. 445, 458

Tomas Kalibera. Replicating real-time garbage collector for Java. In *7th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, Madrid, Spain, September 2009, pages 100–109. ACM Press. doi: 10.1145/1620405.1620420. 405

Tomas Kalibera, Filip Pizlo, Antony L. Hosking, and Jan Vitek. Scheduling hard real-time garbage collection. In *30th IEEE Real-Time Systems Symposium*, Washington, DC, December 2009, pages 81–92. IEEE Computer Society Press. doi: 10.1109/RTSS.2009.40. 415

Haim Kermany and Erez Petrank. The Compressor: Concurrent, incremental and parallel compaction. In PLDI 2006, pages 354–363. doi: 10.1145/1133981.1134023. 38, 39, 202, 302, 352, 361

Taehyoun Kim and Heonshik Shin. Scheduling-aware real-time garbage collection using dual aperiodic servers. In *Real-Time and Embedded Computing Systems and Applications*, 2004, pages 1–17. Volume 2968 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-24686-2_1. 415

Taehyoun Kim, Naehyuck Chang, Namyun Kim, and Heonshik Shin. Scheduling garbage collector for embedded real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Atlanta, GA, May 1999, pages 55–64. ACM SIGPLAN Notices 34(7), ACM Press. doi: 10.1145/314403.314444. 415

Taehyoun Kim, Naehyuck Chang, and Heonshik Shin. Bounding worst case garbage collection time for embedded real-time systems. In *6th IEEE Real-Time Technology and Applications Symposium (RTAS)*, Washington, DC, May/June 2000, pages 46–55. doi: 10.1109/RTTAS.2000.852450. 415

Taehyoun Kim, Naehyuck Chang, and Heonshik Shin. Joint scheduling of garbage collector and hard real-time tasks for embedded applications. *Journal of Systems and Software*, 58(3):247–260, September 2001. doi: 10.1016/S0164-1212(01)00042-5. 415

Andy C. King. *Removing Garbage Collector Synchronisation*. PhD thesis, Computing Laboratory, The University of Kent at Canterbury, 2004. http://www.cs.kent.ac.uk/pubs/2004/1981/. 145, 451

Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, October 1965. doi: 10.1145/365628.365655. 96

Donald E. Knuth. *The Art of Computer Programming*, volume I: Fundamental Algorithms. Addison-Wesley, second edition, 1973. 89, 90, 98

David G. Korn and Kiem-Phong Vo. In search of a better malloc. In *USENIX Summer Conference*, Portland, Oregon, 1985, pages 489–506. USENIX Association. 100, 152

H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, Providence, Rhode Island, October 1977, pages 120–131. IEEE Press. doi: 10.1109/SFCS.1977.5. 326, 329

Michael S. Lam, Paul R. Wilson, and Thomas G. Moher. Object type directed garbage collection to improve locality. In IWMM 1992, pages 404–425. doi: 10.1007/BFb0017204. 52, 53

Leslie Lamport. Garbage collection with multiple processes: an exercise in parallelism. 1976, pages 50–54. IEEE Press. 326, 327

Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *Symposium on Interpreters and Interpretive Techniques*, St Paul, MN, June 1987, pages 253–263. ACM SIGPLAN Notices 22(7), ACM Press. doi: 10.1145/29650.29677. 137, 149, 150, 151, 159

LCTES 2003. *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, San Diego, CA, June 2003. ACM SIGPLAN Notices 38(7), ACM Press. doi: 10.1145/780732. 435, 459

Ho-Fung Leung and Hing-Fung Ting. An optimal algorithm for global termination detection in shared-memory asynchronous multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):538–543, May 1997. doi: 10.1109/71.598280. 248, 249

Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In OOPSLA 2001, pages 367–380. doi: 10.1145/504282.504309. 157, 369

Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems*, 28(1):1–69, January 2006. doi: 10.1145/1111596.1111597. 108, 369, 374

Yossi Levanoni and Erez Petrank. A scalable reference counting garbage collector. Technical Report CS–0967, Technion — Israel Institute of Technology, Haifa, Israel, November 1999. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/1999/CS/CS0967. 63, 331, 369

LFP 1984, Guy L. Steele, editor. *ACM Conference on LISP and Functional Programming*, Austin, TX, August 1984. ACM Press. doi: 10.1145/800055. 439, 447, 454, 462

LFP 1992. *ACM Conference on LISP and Functional Programming*, San Francisco, CA, June 1992. ACM Press. doi: 10.1145/141471. 442, 446

Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983. doi: 10.1145/358141.358147. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981. 103, 116

Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992. doi: 10.1016/0020-0190(92)90088-D. Also Computing Laboratory Technical Report 75, University of Kent, July 1990. 72

Boris Magnusson and Roger Henriksson. Garbage collection for control systems. In IWMM 1995, pages 323–342. doi: 10.1007/3-540-60368-9_32. 386

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *32nd Annual ACM Symposium on Principles of Programming Languages*, Long Beach, CA, January 2005, pages 378–391. ACM SIGPLAN Notices 40(1), ACM Press. doi: 10.1145/1040305.1040336. 346

Sebastien Marion, Richard Jones, and Chris Ryder. Decrypting the Java gene pool: Predicting objects' lifetimes with micro-patterns. In ISMM 2007, pages 67–78. doi: 10.1145/1296907.1296918. 110, 132

Simon Marlow, Tim Harris, Roshan James, and Simon L. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In ISMM 2008, pages 11–20. doi: 10.1145/1375634.1375637. 116, 132, 292, 296, 474

Johannes J. Martin. An efficient garbage compaction algorithm. *Communications of the ACM*, 25(8):571–581, August 1982. doi: 10.1145/358589.358625. 38

A. D. Martinez, R. Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990. doi: 10.1016/0020-0190(90)90226-N. 72

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960. doi: 10.1145/367177.367199. xxiii, 2, 18, 29

John McCarthy. History of LISP. In Richard L. Wexelblat, editor, *History of Programming Languages I*, pages 173–185. ACM Press, 1978. doi: 10.1145/800025.1198360. xxiii

Bill McCloskey, David F. Bacon, Perry Cheng, and David Grove. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. IBM Research Report RC24505, IBM Research, 2008. http://researcher.watson.ibm.com/researcher/files/us-groved/rc24504.pdf. 407, 409

Phil McGachey and Antony L Hosking. Reducing generational copy reserve overhead with fallback compaction. In ISMM 2006, pages 17–28. doi: 10.1145/1133956.1133960. 126

Phil McGachey, Ali-Reza Adl-Tabatabi, Richard L. Hudson, Vijay Menon, Bratin Saha, and Tatiana Shpeisman. Concurrent GC leveraging transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008, pages 217–226. ACM Press. doi: 10.1145/1345206.1345238. 270

Paul E. McKenney and Jack Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *10th IASTED International Conference on Parallel and Distributed Computing and Systems*, October 1998. IEEE Computer Society. 374

Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004. doi: 10.1109/TPDS.2004.8. 374

Maged M. Michael and M.L. Scott. Correction of a memory management method for lock-free data structures. Technical Report UR CSD / TR59, University of Rochester, December 1995. doi: 1802/503. 374

James S. Miller and Guillermo J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report AIM-1462, MIT AI Laboratory, March 1994. doi: 1721.1/6622. 171

David A. Moon. Garbage collection in a large LISP system. In LFP 1984, pages 235–245. doi: 10.1145/800055.802040. 50, 51, 202, 296

F. Lockwood Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–5, 1978. doi: 10.1145/359576.359583. 36, 42, 299

F. Lockwood Morris. On a comparison of garbage collection techniques. *Communications of the ACM*, 22(10):571, October 1979. 37, 42

F. Lockwood Morris. Another compacting garbage collector. *Information Processing Letters*, 15(4):139–142, October 1982. doi: 10.1016/0020-0190(82)90094-1. 37, 38, 42

J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle? *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992. doi: 10.1109/32.153378. 207

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In Mary Lou Soffa, editor, *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, March 2008, pages 265–276. ACM SIGPLAN Notices 43(3), ACM Press. doi: 10.1145/1508244.1508275. 10

John Nagle. Re: Real-time GC (was Re: Widespread C++ competency gap). USENET comp.lang.c++, January 1995. 4

Scott Nettles and James O'Toole. Real-time replication-based garbage collection. In PLDI 1993, pages 217–226. doi: 10.1145/155090.155111. 341, 342, 347, 361, 378

Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In IWMM 1992, pages 357–364. doi: 10.1007/BFb0017201. 341, 361

Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, March 2007, pages 68–78. ACM Press. doi: 10.1145/1229428.1229442. 272

Gene Novark, Trevor Strohman, and Emery D. Berger. Custom object layout for garbage-collected languages. Technical report, University of Massachusetts, 2006. https://web.cs.umass.edu/publication/docs/2006/UM-CS-2006-007.pdf. New England Programming Languages and Systems Symposium, March, 2006. 53

Cosmin E. Oancea, Alan Mycroft, and Stephen M. Watt. A new approach to parallelising tracing algorithms. In ISMM 2009, pages 10–19. doi: 10.1145/1542431.1542434. 262, 263, 264, 265, 298, 304, 305, 474

Takeshi Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Orlando, FL, October 2009, pages 377–390. ACM SIGPLAN Notices 44(10), ACM Press. doi: 10.1145/1640089.1640117. 293, 474

OOPSLA 1999. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, CO, October 1999. ACM SIGPLAN Notices 34(10), ACM Press. doi: 10.1145/320384. 438, 462

OOPSLA 2001. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Tampa, FL, November 2001. ACM SIGPLAN Notices 36(11), ACM Press. doi: 10.1145/504282. 437, 439, 452

OOPSLA 2002. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, November 2002. ACM SIGPLAN Notices 37(11), ACM Press. doi: 10.1145/582419. 460, 466

OOPSLA 2003. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, November 2003. ACM SIGPLAN Notices 38(11), ACM Press. doi: 10.1145/949305. 435, 436, 437, 447, 448, 459

OOPSLA 2004. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 2004. ACM SIGPLAN Notices 39(10), ACM Press. doi: 10.1145/1028976. 433, 435, 447, 449, 459

OOPSLA 2005. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, October 2005. ACM SIGPLAN Notices 40(10), ACM Press. doi: 10.1145/1094811. 446, 448, 465

Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In PLDI 2002, pages 129–140. doi: 10.1145/512529.512546. 284, 285, 288, 296, 304, 480

Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. Mostly concurrent compaction for mark-sweep GC. In ISMM 2004, pages 25–36. doi: 10.1145/1029873.1029877. 321, 352

Ivor P. Page and Jeff Hagins. Improving the performance of buddy systems. *IEEE Transactions on Computers*, C-35(5):441–447, May 1986. doi: 10.1109/TC.1986.1676786. 96

Krzysztof Palacz, Jan Vitek, Grzegorz Czajkowski, and Laurent Daynès. Incommunicado: efficient communication for isolates. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, October 1994, pages 262–274. ACM SIGPLAN Notices 29(10), ACM Press. doi: 10.1145/582419.582444. 107

Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988. doi: 10.1145/63039.6304. 194

Harel Paz and Erez Petrank. Using prefetching to improve reference-counting garbage collectors. In *16th International Conference on Compiler Construction*, Braga, Portugal, March 2007, pages 48–63. Volume 4420 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-71229-9_4. 64

Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. Efficient on-the-fly cycle collection. Technical Report CS–2003–10, Technion University, 2003. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/2003/CS/CS-2003-10. 369, 370

Harel Paz, Erez Petrank, David F. Bacon, Elliot K. Kolodner, and V.T. Rajan. An efficient on-the-fly cycle collection. In CC 2005, pages 156–171. doi: 10.1007/978-3-540-31985-6_11. 369

Harel Paz, Erez Petrank, and Stephen M. Blackburn. Age-oriented concurrent garbage collection. In CC 2005, pages 121–136. doi: 10.1007/978-3-540-31985-6_9. 369

Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. An efficient on-the-fly cycle collection. *ACM Transactions on Programming Languages and Systems*, 29 (4):1–43, August 2007. doi: 10.1145/1255450.1255453. 67, 369, 372, 373

E. J. H. Pepels, M. C. J. D. van Eekelen, and M. J. Plasmeijer. A cyclic reference counting algorithm and its proof. Technical Report 88–10, Computing Science Department, University of Nijmegen, 1988. https://www.cs.ru.nl/~marko/research/pubs/1988/IR88-10CyclicProof-incl-fig.pdf. 67

James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977. doi: 10.1145/359605.359626. 96

Erez Petrank and Elliot K. Kolodner. Parallel copying garbage collection using delayed allocation. *Parallel Processing Letters*, 14(2):271–286, June 2004. doi: 10.1142/S0129626404001878. 284, 289

Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *Twenty-ninth Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002, pages 101–112. ACM SIGPLAN Notices 37(1), ACM Press. doi: 10.1145/503272.503283. 49

Pekka P. Pirinen. Barrier techniques for incremental tracing. In ISMM 1998, pages 20–25. doi: 10.1145/286860.286863. 20, 315, 316, 317, 318

Filip Pizlo and Jan Vitek. Memory management for real-time Java: State of the art. In *11th International Symposium on Object-Oriented Real-Time Distributed Computing*, Orlando, FL, 2008, pages 248–254. IEEE Press. doi: 10.1109/ISORC.2008.40. 377

Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. Stopless: A real-time garbage collector for multiprocessors. In ISMM 2007, pages 159–172. doi: 10.1145/1296907.1296927. 406, 412

Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In PLDI 2008, pages 33–44. doi: 10.1145/1379022.1375587. 410, 411, 412

Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *5th European Conference on Computer Systems (EuroSys)*, Paris, France, April 2010a, pages 69–82. ACM Press. doi: 10.1145/1755913.1755922. 416

Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Canada, June 2010b, pages 146–159. ACM SIGPLAN Notices 45(6), ACM Press. doi: 10.1145/1806596.1806615. 413, 414, 415, 416

PLDI 1988. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, June 1988. ACM SIGPLAN Notices 23(7), ACM Press. doi: 10.1145/53990. 434, 440

PLDI 1991. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991. ACM SIGPLAN Notices 26(6), ACM Press. doi: 10.1145/113445. 439, 446, 465

PLDI 1992. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992. ACM SIGPLAN Notices 27(7), ACM Press. doi: 10.1145/143095. 434, 443

PLDI 1993. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993. ACM SIGPLAN Notices 28(6), ACM Press. doi: 10.1145/155090. 436, 438, 444, 455

PLDI 1997. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997. ACM SIGPLAN Notices 32(5), ACM Press. doi: 10.1145/258915. 442, 443

PLDI 1999. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999. ACM SIGPLAN Notices 34(5), ACM Press. doi: 10.1145/301618. 438, 441, 462

PLDI 2000. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000. ACM SIGPLAN Notices 35(5), ACM Press. doi: 10.1145/349299. 439, 443, 444, 459

PLDI 2001. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, UT, June 2001. ACM SIGPLAN Notices 36(5), ACM Press. doi: 10.1145/378795. 434, 435, 441

PLDI 2002. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002. ACM SIGPLAN Notices 37(5), ACM Press. doi: 10.1145/512529. 437, 446, 455

PLDI 2006, Michael I. Schwartzbach and Thomas Ball, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006. ACM SIGPLAN Notices 41(6), ACM Press. doi: 10.1145/1133981. 441, 451, 464

PLDI 2008, Rajiv Gupta and Saman P. Amarasinghe, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, AZ, June 2008. ACM SIGPLAN Notices 43(6), ACM Press. doi: 10.1145/1375581. 437, 438, 457

POPL 1994. *21st Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 1994. ACM Press. doi: 10.1145/174675. 443, 444, 463

POPL 2003. *30th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, LA, January 2003. ACM SIGPLAN Notices 38(1), ACM Press. doi: 10.1145/604131. 435, 438

POS 1992, Antonio Albano and Ronald Morrison, editors. *5th International Workshop on Persistent Object Systems (September, 1992)*, San Miniato, Italy, 1992. Workshops in Computing, Springer. doi: 10.1007/978-1-4471-3209-7. 461, 466

Tony Printezis. Hot-Swapping between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment. In JVM 2001. http://www.usenix.org/events/jvm01/printezis.html. 6, 41, 138

Tony Printezis. On measuring garbage collection responsiveness. *Science of Computer Programming*, 62(2):164–183, October 2006. doi: 10.1016/j.scico.2006.02.004. 375, 376, 415

Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In ISMM 2000, pages 143–154. doi: 10.1145/362422.362480. 23, 151, 326

Tony Printezis and Alex Garthwaite. Visualising the Train garbage collector. In ISMM 2002, pages 100–105. doi: 10.1145/512429.512436. 22, 73, 143

Feng Qian and Laurie Hendren. An adaptive, region-based allocator for Java. In ISMM 2002, pages 127–138. doi: 10.1145/512429.512446. Sable Technical Report 2002–1 provides a longer version. 147

Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark DURING Sweep rather than Mark THEN Sweep. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *Parallel Architectures and Languages Europe (PARLE)*, Eindhoven, The Netherlands, June 1989, pages 224–237. Volume 365 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3540512845_42. 328

John H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, December 1993. http://www.smlnj.org/compiler-notes/93-tr-reppy.ps. 105, 121, 195, 201

Sven Gestegøard Robertz and Roger Henriksson. Time-triggered garbage collection: Robust and adaptive real-time GC scheduling for embedded systems. In LCTES 2003, pages 93–102. doi: 10.1145/780732.780745. 415

J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, 18(3):416–423, July 1971. doi: 10.1145/321650.321658. 30

J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):419–499, July 1974. doi: 10.1145/321832.321846. 30

J. M. Robson. A bounded storage algorithm for copying cyclic structures. *Communications of the ACM*, 20(6):431–433, June 1977. doi: 10.1145/359605.359628. 90

J. M. Robson. Storage allocation is NP-hard. *Information Processing Letters*, 11(3):119–125, November 1980. doi: 10.1016/0020-0190(80)90124-6. 93

Helena C. C. D. Rodrigues and Richard E. Jones. Cyclic distributed garbage collection with group merger. In Eric Jul, editor, *12th European Conference on Object-Oriented Programming*, Brussels, Belgium, July 1998, pages 249–273. Volume 1445 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/BFb0054095. Also UKC Technical report 17–97, December 1997. 58

Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language. Technical Report CSL–84–7, Xerox PARC, Palo Alto, CA, July 1985. http://www.bitsavers.org/pdf/xerox/parc/techReports/CSL-84-7_On_Adding_Garbage_Collection_and_Runtime_Types_to_a_Strongly-Typed_Statically-Checked_Concurrent_Language.pdf. 4

Erik Ruf. Effective synchronization removal for Java. In PLDI 2000, pages 208–218. doi: 10.1145/349299.349327. 145

Narendran Sachindran and Eliot Moss. MarkCopy: Fast copying GC with less space overhead. In OOPSLA 2003, pages 326–343. doi: 10.1145/949305.949335. 154, 155, 159

Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. MC^2: High-performance garbage collection for memory-constrained environments. In OOPSLA 2004, pages 81–98. doi: 10.1145/1028976.1028984. 7, 155, 159

Konstantinos Sagonas and Jesper Wilhelmsson. Message analysis-guided allocation and low-pause incremental garbage collection in a concurrent language. In ISMM 2004, pages 1–12. doi: 10.1145/1029873.1029875. 146

Konstantinos Sagonas and Jesper Wilhelmsson. Efficient memory management for concurrent programs that use message passing. *Science of Computer Programming*, 62(2): 98–121, October 2006. doi: 10.1016/j.scico.2006.02.006. 146

Jon D. Salkild. Implementation and analysis of two reference counting algorithms. Master's thesis, University College, London, 1987. 67

Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. In John Hughes, editor, *Conference on Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993, pages 106–116. ACM Press. doi: 10.1145/165180.165195. 113

Robert A. Saunders. The LISP system for the Q–32 computer. In E. C. Berkeley and Daniel G. Bobrow, editors, *The Programming Language LISP: Its Operation and Applications*, Cambridge, MA, 1974, pages 220–231. Information International, Inc. http://www.softwarepreservation.org/projects/LISP/book/III_LispBook_Apr66.pdf. 32

Martin Schoeberl. Scheduling of hard real-time garbage collection. *Real-Time Systems*, 45 (3):176–213, 2010. doi: 10.1007/s11241-010-9095-4. 415

Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In Oscar Nierstrasz, editor, *9th European Conference on Object-Oriented Programming*, øAarhus, Denmark, August 1995, pages 235–252. Volume 952 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-49538-X_12. 143

Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Stanford University, 1988. Technical Report CSL-TR-88-351. 116, 118, 192, 202

Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In OOPSLA 2002, pages 13–25. doi: 10.1145/582419.582422. 53

Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, CA, November 2000, pages 9–17. ACM Press. doi: 10.1145/354880.354883. 385, 412, 415

Fridtjof Siebert. Limits of parallel marking collection. In ISMM 2008, pages 21–29. doi: 10.1145/1375634.1375638. 276, 277, 303, 480

Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. In ISMM 2010, pages 11–20. doi: 10.1145/1806651.1806654. 280, 282, 283, 304, 385, 412, 480

Fridtjof Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In ISMM 1998, pages 130–137. doi: 10.1145/286860.286874. 385, 412

Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *6th International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, Hong Kong, 1999, pages 96–102. IEEE Press, IEEE Computer Society Press. doi: 10.1109/RTCSA.1999.811198. 182

David Siegwart and Martin Hirzel. Improving locality with parallel hierarchical copying GC. In ISMM 2006, pages 52–63. doi: 10.1145/1133956.1133964. 52, 295, 296, 297, 304, 474

Jeremy Singer, Gavin Brown, Mikel Lujan, and Ian Watson. Towards intelligent analysis techniques for object pretenuring. In *ACM International Symposium on Principles and Practice of Programming in Java*, Lisbon, Portugal, September 2007a, pages 203–208. Volume 272 of *ACM International Conference Proceeding Series*. doi: 10.1145/1294325.1294353. 80

Jeremy Singer, Gavin Brown, Ian Watson, and John Cavazos. Intelligent selection of application-specific garbage collectors. In ISMM 2007, pages 91–102. doi: 10.1145/1296907.1296920. 6

Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: an efficient, portable persistent store. In POS 1992, pages 11–33. doi: 10.1007/978-1-4471-3209-7. 207

Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):562–686, July 1985. doi: 10.1145/3828.3835. 92

Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Bachelor of Science thesis AITR-1417, MIT AI Lab, February 1988. doi: 1721.1/6795. 197

Sunil Soman and Chandra Krintz. Efficient and general on-stack replacement for aggressive program specialization. In *International Conference on Software Engineering Research and Practice (SERP) & Conference on Programming Languages and Compilers, Volume 2*, Las Vegas, NV, June 2006, pages 925–932. CSREA Press. https://www.cs.ucsb.edu/~ckrintz/papers/osr.pdf. 190

Sunil Soman, Chandra Krintz, and David Bacon. Dynamic selection of application-specific garbage collectors. In ISMM 2004, pages 49–60. doi: 10.1145/1029873.1029880. 6, 41, 80

Sunil Soman, Laurent Daynès, and Chandra Krintz. Task-aware garbage collection in a multi-tasking virtual machine. In ISMM 2006, pages 64–73. doi: 10.1145/1133956.1133965. 107

Sunil Soman, Chandra Krintz, and Laurent Daynès. MTM^2: Scalable memory management for multi-tasking managed runtime environments. In Jan Vitek, editor, *22nd European Conference on Object-Oriented Programming*, Paphos, Cyprus, July 2008, pages 335–361. Volume 5142 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-70592-5_15. 107

Daniel Spoonhower, Guy Blelloch, and Robert Harper. Using page residency to balance tradeoffs in tracing garbage collection. In VEE 2005, pages 57–67. doi: 10.1145/1064979.1064989. 149, 150, 152

James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(3):155–180, May 1984. doi: 10.1145/190.194. 50

James William Stamos. A large object-oriented virtual memory: Grouping strategies, measurements, and performance. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, April 1982. doi: 1721.1/15807. 50

Thomas A. Standish. *Data Structure Techniques*. Addison-Wesley, 1980. 87, 92

Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975. doi: 10.1145/361002.361005. 229, 315, 316, 330

Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976. doi: 10.1145/360238.360247. 315, 316, 318, 323, 326, 335

Peter Steenkiste. *Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization*. PhD thesis, Stanford University, March 1987. Available as Technical Report CSL-TR-87-324. 27

Peter Steenkiste. The impact of code density on instruction cache performance. In *16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989, pages 252–259. IEEE Press. doi: 10.1145/74925.74954. 80

Peter Steenkiste and John Hennessy. Lisp on a reduced-instruction-set processor: Characterization and optimization. *IEEE Computer*, 21(7):34–45, July 1988. doi: 10.1109/2.67. 27

Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In ISMM 2000, pages 18–24. doi: 10.1145/362422.362432. 107, 145, 146, 159

Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999. 128, 157

Darko Stefanović and J. Eliot B. Moss. Characterisation of object behaviour in Standard ML of New Jersey. In *ACM Conference on LISP and Functional Programming*, Orlando, FL, June 1994, pages 43–54. ACM Press. doi: 10.1145/182409.182428. 113

Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In OOPSLA 1999, pages 370–381. doi: 10.1145/320384.320425. 128

Darko Stefanović, Matthew Hertz, Stephen Blackburn, Kathryn McKinley, and J. Eliot Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Workshop on Memory System Performance*, Berlin, Germany, June 2002, pages 25–36. ACM SIGPLAN Notices 38(2 supplement), ACM Press. doi: 10.1145/773146.773042. 129

V. Stenning. On-the-fly garbage collection. Unpublished notes, cited by Gries [1977], 1976. 315

C. J. Stephenson. New methods of dynamic storage allocation (fast fits). In *9th ACM Symposium on Operating Systems Principles*, Bretton Woods, NH, October 1983, pages 30–32. ACM SIGOPS Operating Systems Review 17(5), ACM Press. doi: 10.1145/800217.806613. 92

James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. Support for garbage collection at every instruction in a Java compiler. In PLDI 1999, pages 118–127. doi: 10.1145/301618.301652. 179, 180, 181, 188

Will R. Stoye, T. J. W. Clarke, and Arthur C. Norman. Some practical methods for rapid combinator reduction. In LFP 1984, pages 159–166. doi: 10.1145/800055.802032. 73

Sun Microsystems. Memory management in the Java HotSpot Virtual Machine, April 2006. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf. Technical White Paper. 41, 120

H. Sundell. Wait-free reference counting and memory management. In *19th International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, April 2005. IEEE Computer Society Press. doi: 10.1109/IPDPS.2005.451. 374

Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005. http://www.drdobbs.com/web-development/a-fundamental-turn-toward-concurrency-in/184405990. 275

M. Swanson. An improved portable copying garbage collector. OPnote 86–03, University of Utah, February 1986. 192

M. Tadman. Fast-fit: A new hierarchical dynamic storage allocation technique. Master's thesis, University of California, Irvine, 1978. 92

David Tarditi. Compact garbage collection tables. In ISMM 2000, pages 50–58. doi: 10.1145/362422.362437. 179, 180, 181, 182

Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. In ISMM 2011, pages 79–88. doi: 10.1145/1993478.1993491. 355

S. P. Thomas, W. T. Charnell, S. Darnell, B. A. A. Dias, P. J. Guthrie, J. P. Kramskoy, J. J. Sexton, M. J. Wynn, K. Rautenbach, and W. Plummer. Low-contention grey object sets for concurrent, marking garbage collection. United States Patent 6925637, 1998. 285, 304

Stephen P. Thomas. *The Pragmatics of Closure Reduction*. PhD thesis, The Computing Laboratory, University of Kent at Canterbury, October 1993. 170

Stephen P. Thomas. Having your cake and eating it: Recursive depth-first copying garbage collection with no extra stack. Personal communication, May 1995a. 170

Stephen P. Thomas. Garbage collection in shared-environment closure reducers: Space-efficient depth first copying using a tailored approach. *Information Processing Letters*, 56(1):1–7, October 1995b. doi: 10.1016/0020-0190(95)00131-U. 170

Stephen P. Thomas and Richard E. Jones. Garbage collection for shared environment closure reducers. Technical Report 31–94, University of Kent and University of Nottingham, December 1994. http://www.cs.kent.ac.uk/pubs/1994/147/. 170, 190

Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In POPL 1994, pages 188–201. doi: 10.1145/174675.177855. xxv, 106

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3): 245–265, September 2004. doi: 10.1023/B:LISP.0000029446.78563.a4. 148, 159

Tomoharu, Richard Jones, and Carl G. Ritson. Reference object processing in on-the-fly garbage collection. In Samuel Z. Guyer and David Grove, editors, *13th International Symposium on Memory Management*, Edinburgh, June 2014, pages 59–69. ACM Press. doi: 10.1145/2602988.2602991. 223

David A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9:31–49, January 1979. doi: 10.1002/spe.4380090105. 66

David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984, pages 157–167. ACM SIGPLAN Notices 19(5), ACM Press. doi: 10.1145/800020.808261. 61, 63, 103, 106, 116, 119, 120, 130

David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM distinguished dissertation 1986. MIT Press, 1986. 114

David M. Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, November 1988, pages 1–17. ACM SIGPLAN Notices 23(11), ACM Press. doi: 10.1145/62083.62085. 114, 116, 121, 123, 137, 138, 140

David M. Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992. doi: 10.1145/111186.116734. 121, 123, 138

Maxime van Assche, Joël Goossens, and Raymond R. Devillers. Joint garbage collection and hard real-time scheduling. *Journal of Embedded Computing*, 2(3–4):313–326, 2006. http://content.iospress.com/articles/journal-of-embedded-computing/jec00070. Also published in RTS'05 International Conference on Real-Time Systems, 2005. 415

Martin Vechev. *Derivation and Evaluation of Concurrent Collectors*. PhD thesis, University of Cambridge, 2007. 331

Martin Vechev, David F. Bacon, Perry Cheng, and David Grove. Derivation and evaluation of concurrent collectors. In Andrew P. Black, editor, *19th European Conference on Object-Oriented Programming*, Glasgow, Scotland, July 2005, pages 577–601. Volume 3586 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/11531142_25. 311, 331

Martin T. Vechev, Eran Yahav, and David F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In PLDI 2006, pages 341–353. doi: 10.1145/1133981.1134022. 326, 331

Martin T. Vechev, Eran Yahav, David F. Bacon, and Noam Rinetzky. CGCExplorer: A semi-automated search procedure for provably correct concurrent collectors. In Jeanne Ferrante and Kathryn S. McKinley, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007, pages 456–467. ACM SIGPLAN Notices 42(6), ACM Press. doi: 10.1145/1250734.1250787. 313

VEE 2005, Michael Hind and Jan Vitek, editors. *1st ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Chicago, IL, June 2005. ACM Press. doi: 10.1145/1064979. 441, 445, 461

David Vengerov. Modeling, analysis and throughput optimization of a generational garbage collector. In ISMM 2009, pages 1–9. doi: 10.1145/1542431.1542433. 123

Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 29(4): 229–239, April 1980. doi: 10.1145/358841.358852. 92

Michal Wegiel and Chandra Krintz. The mapping collector: Virtual memory support for generational, parallel, and concurrent compaction. In Susan J. Eggers and James R. Larus, editors, *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, March 2008, pages 91–102. ACM SIGPLAN Notices 43(3), ACM Press. doi: 10.1145/1346281.1346294. 107

J. Weizenbaum. Recovery of reentrant list structures in SLIP. *Communications of the ACM*, 12(7):370–372, July 1969. doi: 10.1145/363156.363159. 60

Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In Martin Odersky, editor, *18th European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004, pages 519–542. Volume 3086 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-24851-4_24. 272

Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Safe futures for Java. In OOPSLA 2005, pages 439–453. doi: 10.1145/1094811.1094845. 272

Derek White and Alex Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, December 1998. http://dl.acm.org/ft_gateway.cfm?id=974971&type=pdf. 118

Jon L. White. Address/memory management for a gigantic Lisp environment, or, GC Considered Harmful. In *LISP Conference*, Stanford University, CA, August 1980, pages 119–127. ACM Press. doi: 10.1145/800087.802797. 49, 107

Jon L. White. Three issues in objected-oriented garbage collection. In GC 1990. ftp://ftp.cs.utexas.edu/pub/garbage/GC90/White.ps.Z. 139

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenstrøm. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computer Systems*, 7(3), April 2008. doi: 10.1145/1347375.1347389. 415

Jesper Wilhelmsson. *Efficient Memory Management for Message-Passing Concurrency — part I: Single-threaded execution*. Licentiate thesis, Uppsala University, May 2005. 146

Paul R. Wilson. A simple bucket-brigade advancement mechanism for generation-based garbage collection. *ACM SIGPLAN Notices*, 24(5):38–46, May 1989. doi: 10.1145/66068.66070. 116

Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps. Expanded version of the IWMM92 paper. 3, 310, 312, 314

Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In GC 1993. ftp://ftp.cs.utexas.edu/pub/garbage/GC93/wilson.ps. 139

Paul R. Wilson and Thomas G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, 1989a. doi: 10.1145/66068.66077. 197

Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New Orleans, LA, October 1989b, pages 23–35. ACM SIGPLAN Notices 24(10), ACM Press. doi: 10.1145/74877.74882. 116, 118, 120, 121, 197

Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In PLDI 1991, pages 177–191. doi: 10.1145/113445.113461. 50, 51, 53, 296

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In IWMM 1995, pages 1–116. doi: 10.1007/3-540-60368-9_19. 10, 90

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Memory allocation policies reconsidered. ftp://ftp.cs.utexas.edu/pub/garbage/submit/PUT_IT_HERE/frag.ps. Unpublished manuscript, 1995b. 96

David S. Wise. The double buddy-system. Computer Science Technical Report TR79, Indiana University, Bloomington, IN, December 1978. https://www.cs.indiana.edu/ftp/techreports/TR79.pdf. 96

David S. Wise. Stop-and-copy and one-bit reference counting. Computer Science Technical Report 360, Indiana University, March 1993a. https://www.cs.indiana.edu/ftp/techreports/TR360.pdf. See also Wise [1993b]. 73

David S. Wise. Stop-and-copy and one-bit reference counting. *Information Processing Letters*, 46(5):243–249, July 1993b. doi: 10.1016/0020-0190(93)90103-G. 466

David S. Wise and Daniel P. Friedman. The one-bit reference count. *BIT*, 17(3):351–359, 1977. doi: 10.1007/BF01932156. 73

P. Tucker Withington. How real is "real time" garbage collection? In GC 1991. ftp://ftp.cs.utexas.edu/pub/garbage/GC91/withington.ps. 104, 140

Mario I. Wolczko and Ifor Williams. Multi-level garbage collection in a high-performance persistent object system. In POS 1992, pages 396–418. doi: 10.1007/978-1-4471-3209-7. 108

Ming Wu and Xiao-Feng Li. Task-pushing: a scalable parallel GC marking algorithm without synchronization operations. In *IEEE International Parallel and Distribution Processing Symposium (IPDPS)*, Long Beach, CA, March 2007, pages 1–10. doi: 10.1109/IPDPS.2007.370317. 288, 289, 298, 304, 480

Feng Xian, Witawas Srisa-an, C. Jia, and Hong Jiang. AS-GC: An efficient generational garbage collector for Java application servers. In ECOOP 2007, pages 126–150. doi: 10.1007/978-3-540-73589-2_7. 107

Ting Yang, Emery D. Berger, Matthew Hertz, Scott F. Kaplan, and J. Eliot B. Moss. Autonomic heap sizing: Taking real memory into account. In ISMM 2004, pages 61–72. doi: 10.1145/1029873.1029881. 209

Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, March 1990. doi: 10.1016/0164-1212(90)90084-Y. 316, 317, 318, 323, 326, 378, 472

Karen Zee and Martin Rinard. Write barrier removal by static analysis. In OOPSLA 2002, pages 191–210. doi: 10.1145/582419.582439. 132, 143, 163, 323

Chengliang Zhang, Kirk Kelsey, Xipeng Shen, Chen Ding, Matthew Hertz, and Mitsunori Ogihara. Program-level adaptive memory management. In ISMM 2006, pages 174–183. doi: 10.1145/1133956.1133979. 210

W. Zhao, K. Ramamritham, and J. A. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Transactions on Software Engineering*, SE-13 (5):564–577, May 1987. doi: 10.1109/TSE.1987.233201. 415

Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, November 1990. http://scholar.colorado.edu/csci_techreports/475/. 124, 125, 323, 393

Benjamin Zorn. The measured cost of conservative garbage collection. *Software: Practice and Experience*, 23:733–756, 1993. doi: 10.1002/spe.4380230704. 116

Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California, Berkeley, March 1989. http://www.eecs.berkeley.edu/Pubs/TechRpts/1989/CSD-89-544.pdf. Technical Report UCB/CSD 89/544. 10, 113