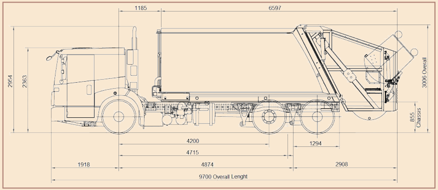
Garbage collection



Jeremy Yallop

jeremy.yallop@cl.cam.ac.uk

Algorithms

Vocabulary

A **heap**: of one or more blocks of contiguous words

A **object**: a heap-allocated contiguous region addressed by 0+ pointers

A **mutator**: application thread, opaque to the collector except for heap operations (allocate, read, write)

A **root**: a heap pointer accessible to the mutator (e.g. in static global storage, stack space, or registers)

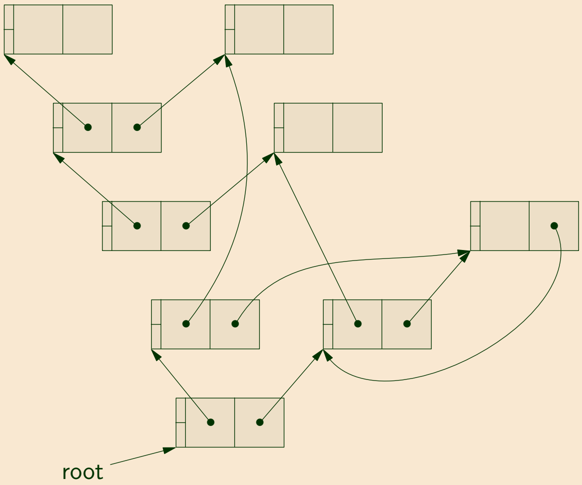
An object is **live** if a mutator will access it in the future

An object is **reachable** if there is a chain of pointers to it from a root

Mark-and-sweep collection

Mark

```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



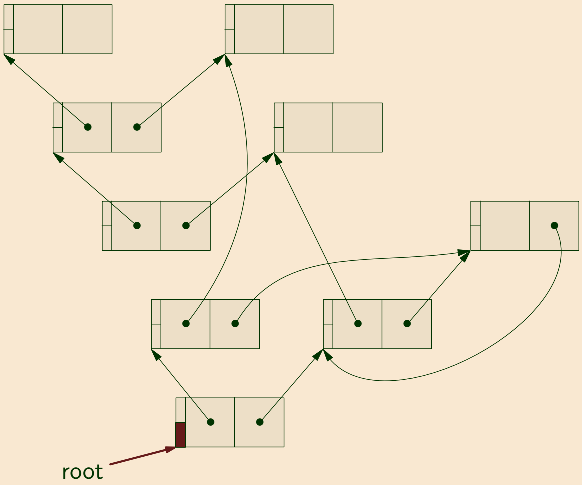
Algorithms

Performance

Reading

Mark

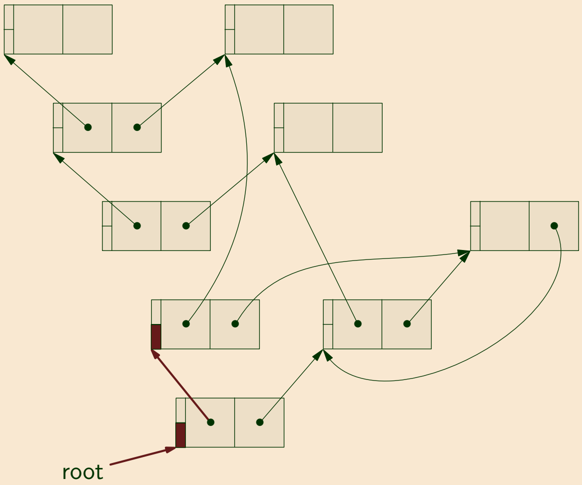
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark-and-sweep collection

Mark

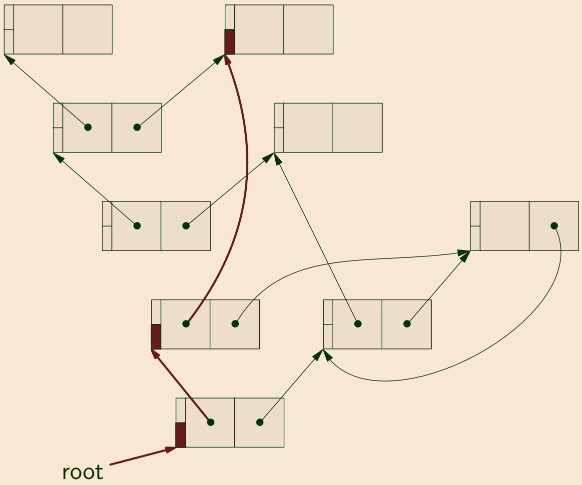
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark-and-sweep collection

Mark

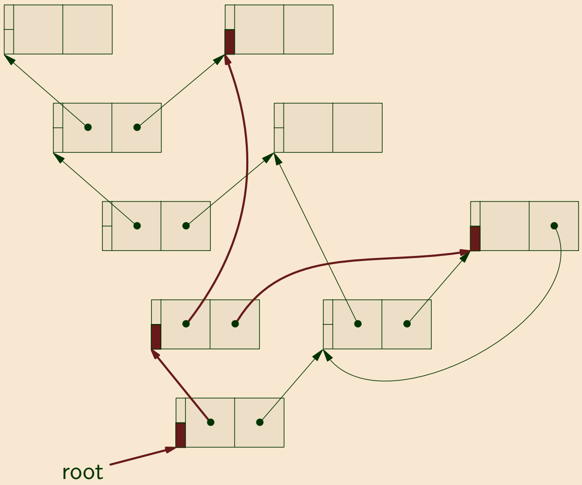
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark-and-sweep collection

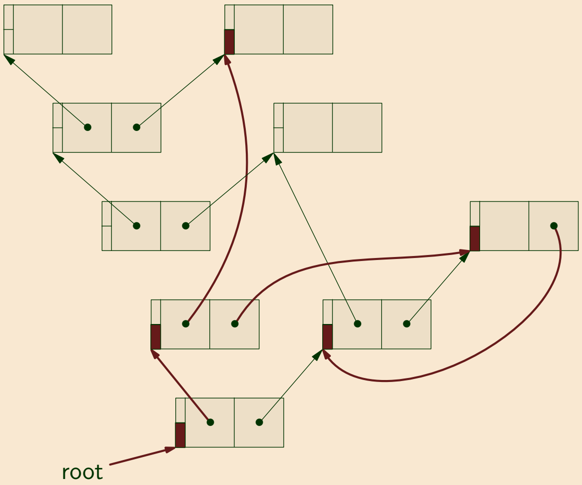
Mark

```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark

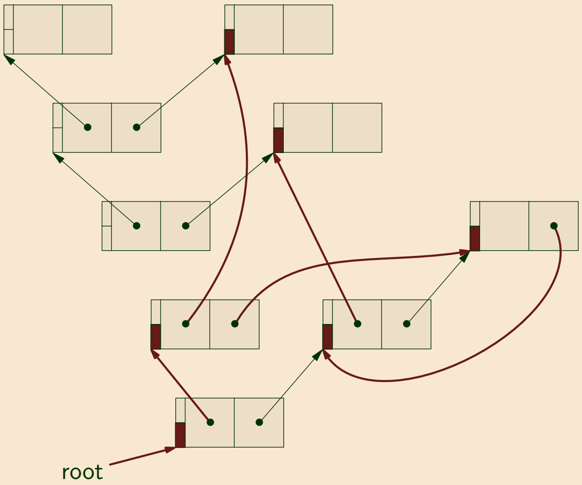
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark-and-sweep collection

Mark

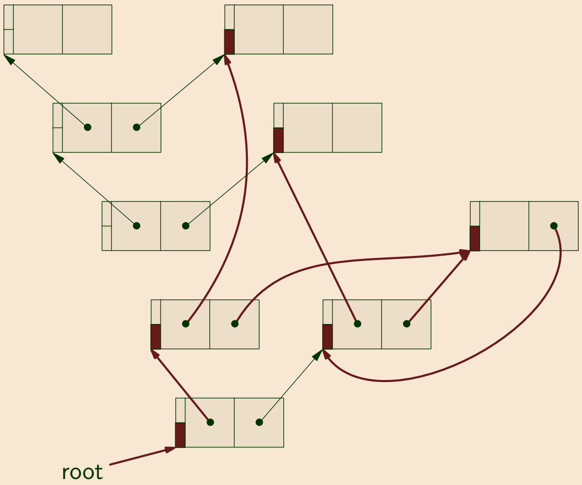
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark-and-sweep collection

Mark

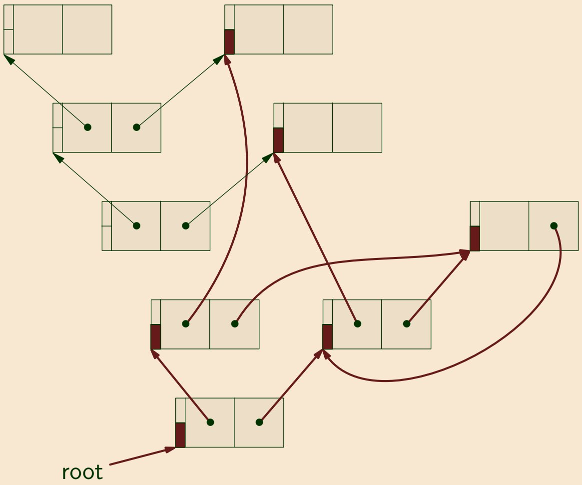
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark-and-sweep collection

Mark

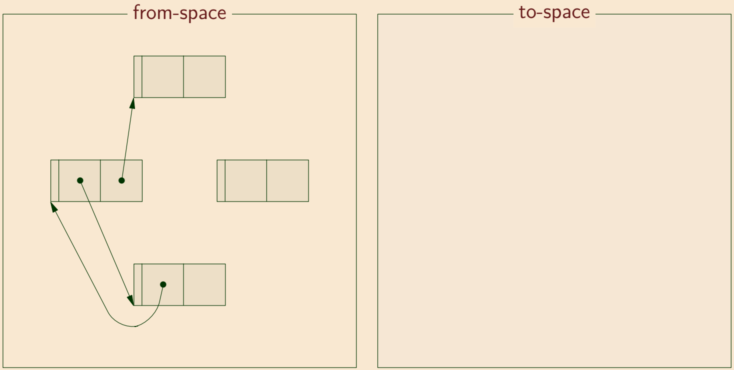
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Copying collection

Collect

copy live blocks to to-space (starting at the root)
leave **forwarding addresses** in from-space
switch roles of spaces



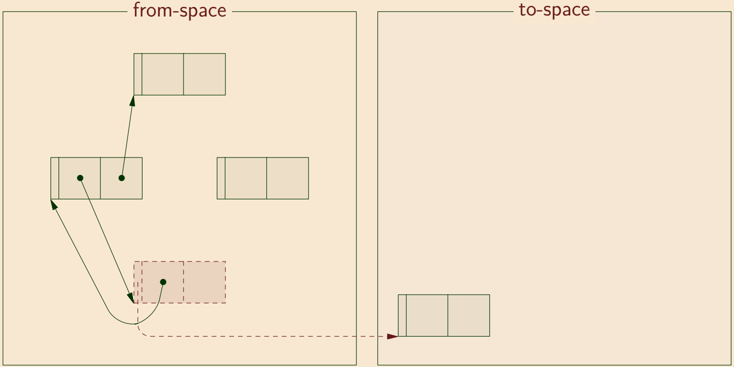
Copying collection

Collect

copy live blocks to to-space (starting at the root)

leave **forwarding addresses** in from-space

switch roles of spaces



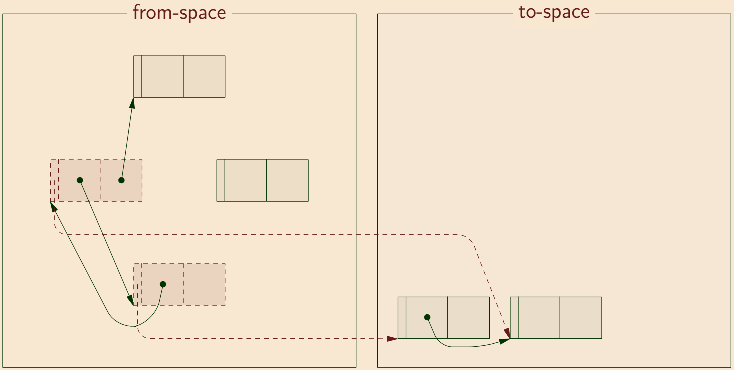
Copying collection

Collect

copy live blocks to to-space (starting at the root)

leave **forwarding addresses** in from-space

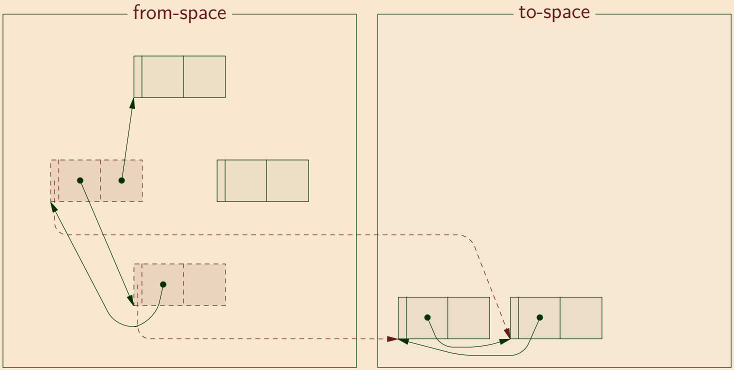
switch roles of spaces



Copying collection

Collect

copy live blocks to to-space (starting at the root)
leave **forwarding addresses** in from-space
switch roles of spaces



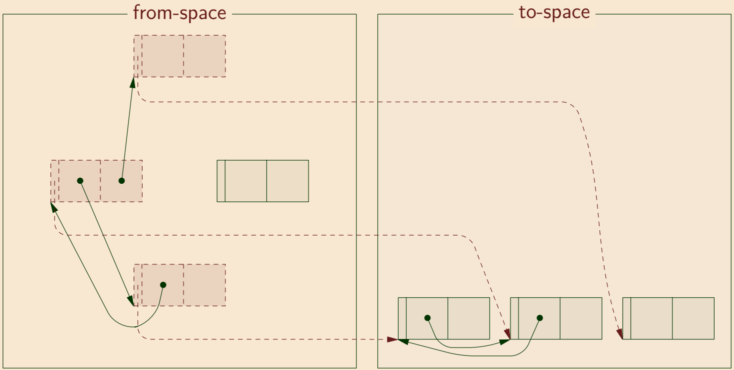
Copying collection

Collect

copy live blocks to to-space (starting at the root)

leave **forwarding addresses** in from-space

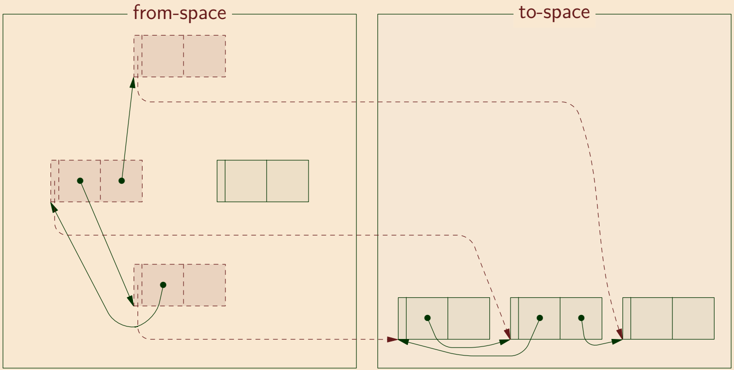
switch roles of spaces



Copying collection

Collect

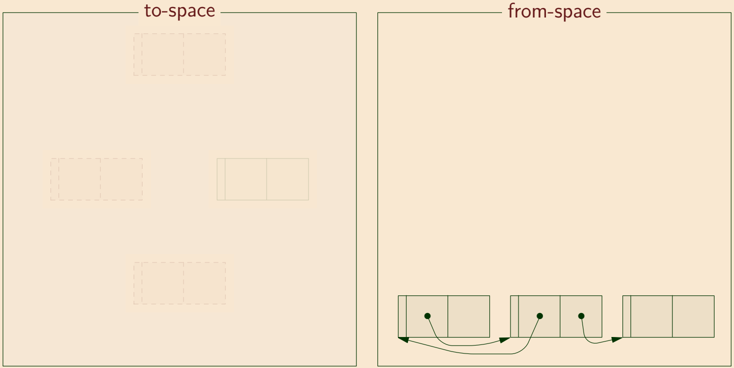
copy live blocks to to-space (starting at the root)
leave **forwarding addresses** in from-space
switch roles of spaces



Copying collection

Collect

copy live blocks to to-space (starting at the root)
leave **forwarding addresses** in from-space
switch roles of spaces



Reference counting

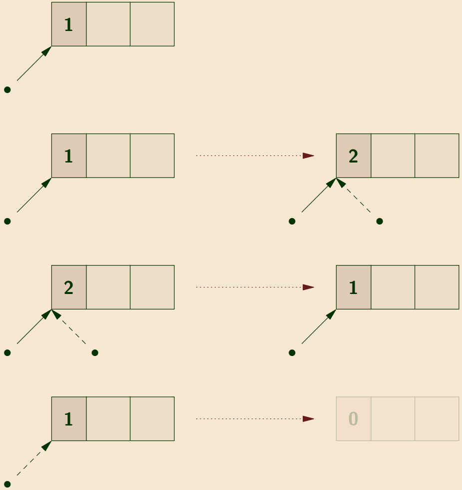
The **reference count** tracks the number of pointers to each object.

An object's reference count is 1 when the object is created:

The count is incremented when a pointer newly references the object:

The count is decremented when a pointer no longer references the object:

The object is unreachable garbage when the reference count goes to 0:



Conservative collection

Motivation: collector has imperfect information about object layout (e.g. because language is compiled to C)

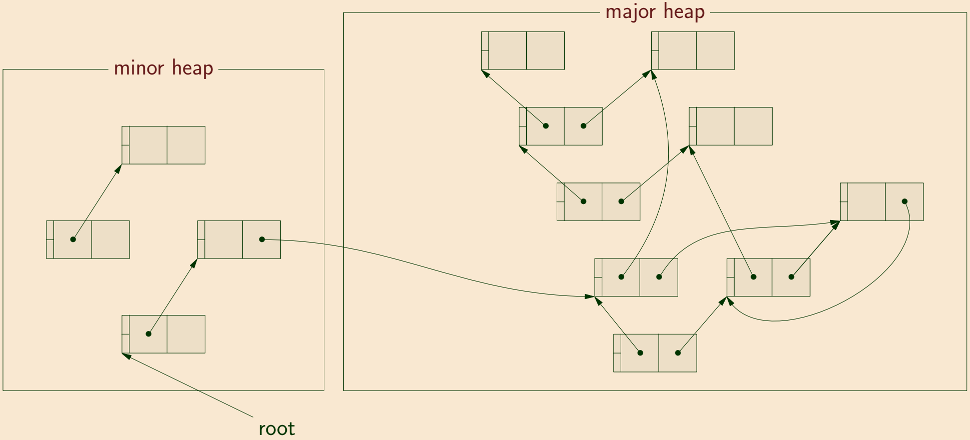
Idea: use an approximation to guess whether a value represents a pointer, e.g.:

1. does the value point into the heap?
2. does it point to valid metadata?

Drawbacks

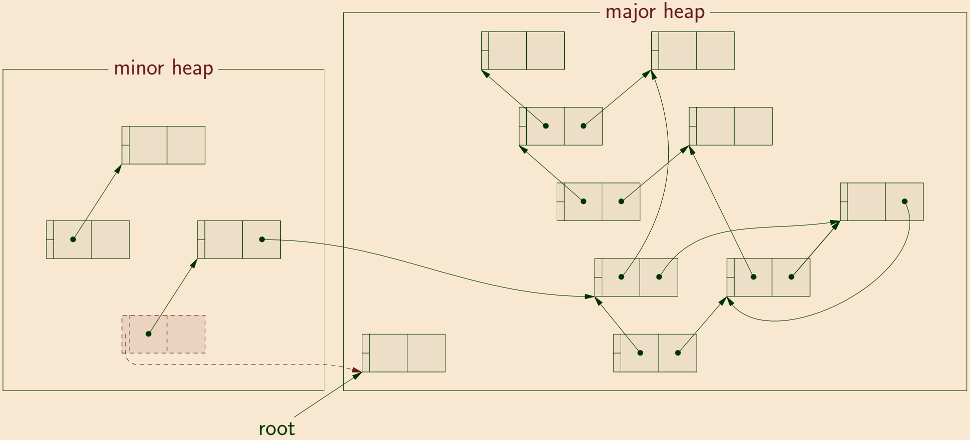
1. (chance) can incorrectly classify addresses as pointers
2. (subterfuge) can fail to identify disguised pointers

Copying collector for minor heap / mark-and-sweep for major heap

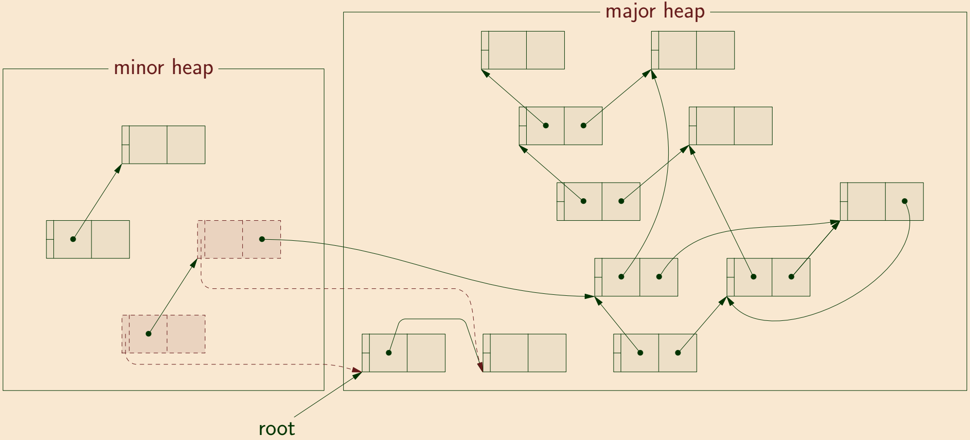


Generational collection

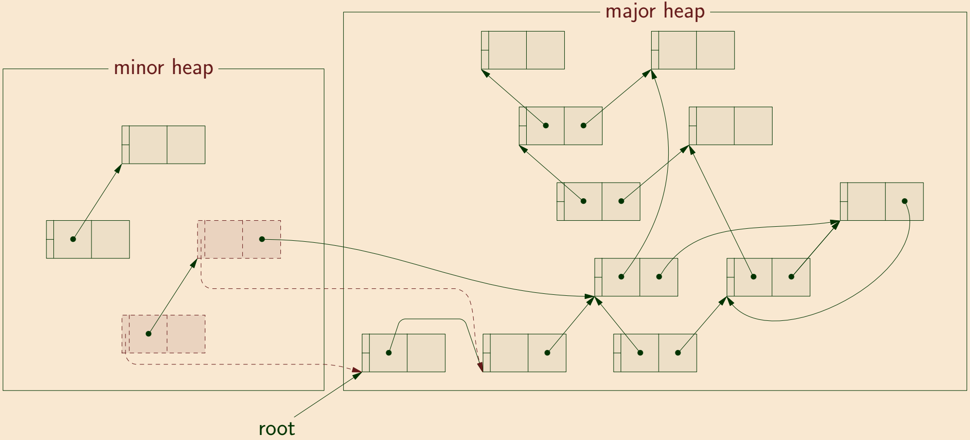
Copying collector for minor heap / mark-and-sweep for major heap



Copying collector for minor heap / mark-and-sweep for major heap

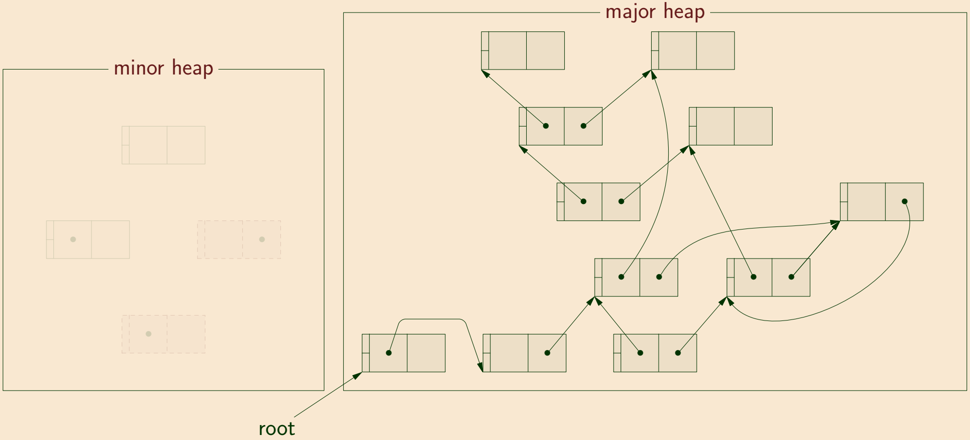


Copying collector for minor heap / mark-and-sweep for major heap

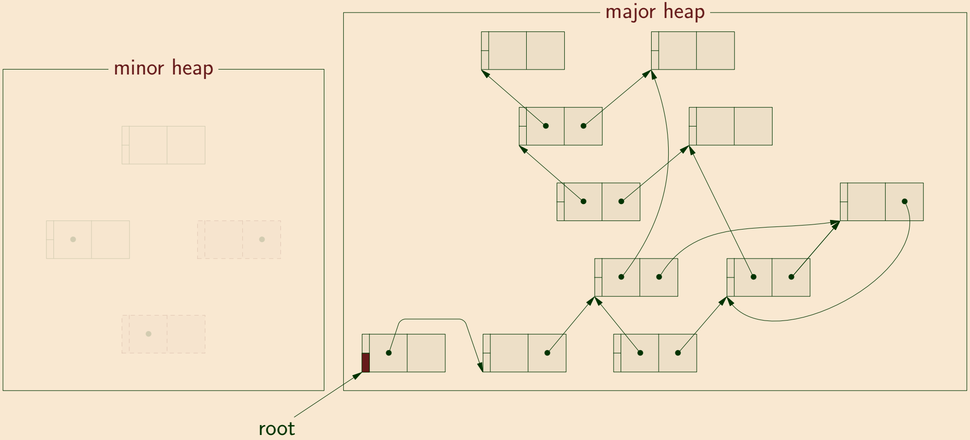


Generational collection

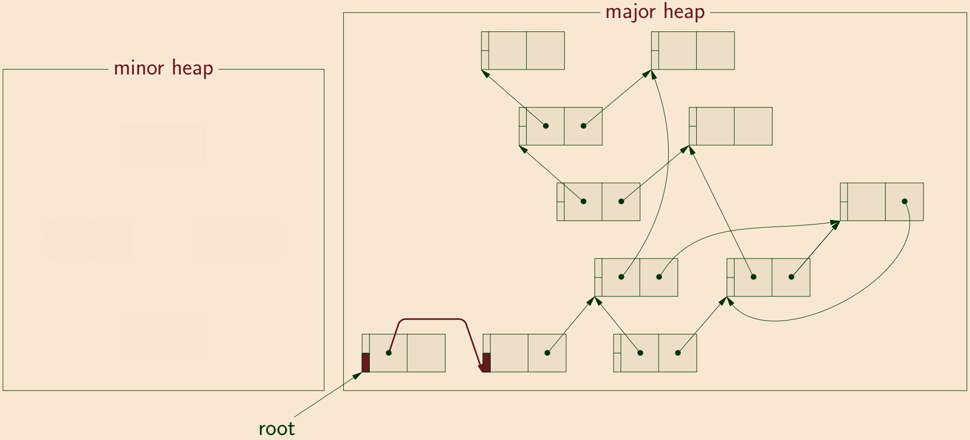
Copying collector for minor heap / mark-and-sweep for major heap



Copying collector for minor heap / mark-and-sweep for major heap

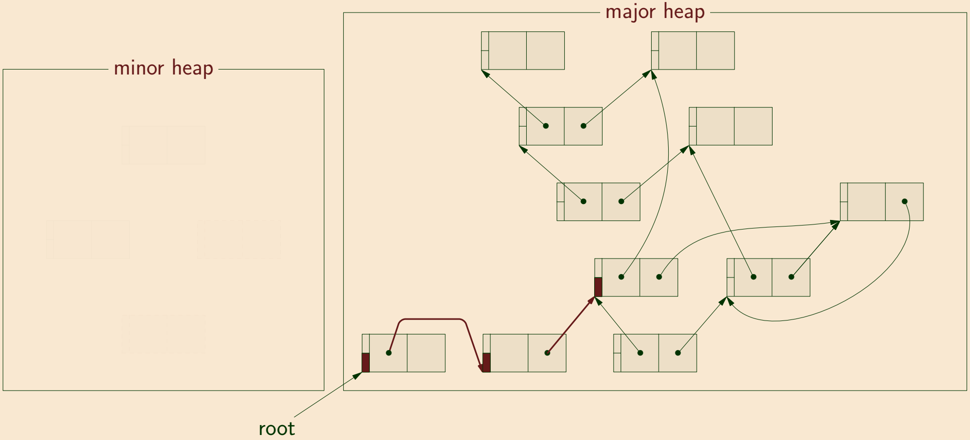


Copying collector for minor heap / mark-and-sweep for major heap

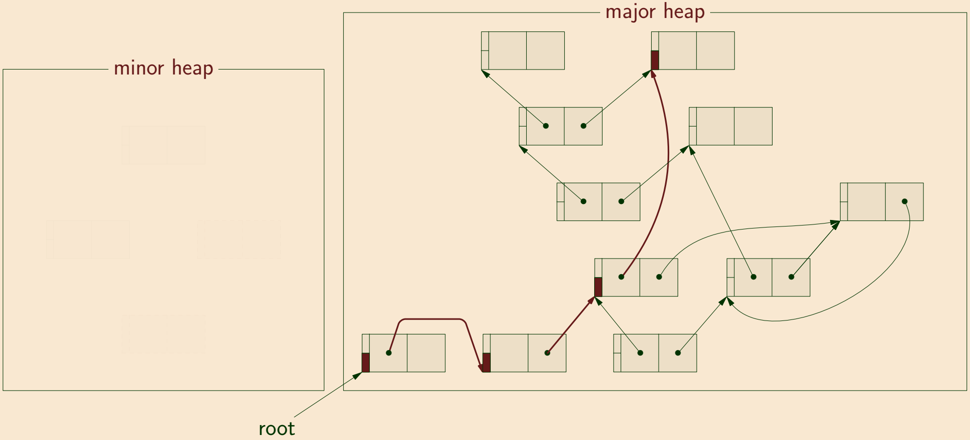


Algorithms
Performance
Reading

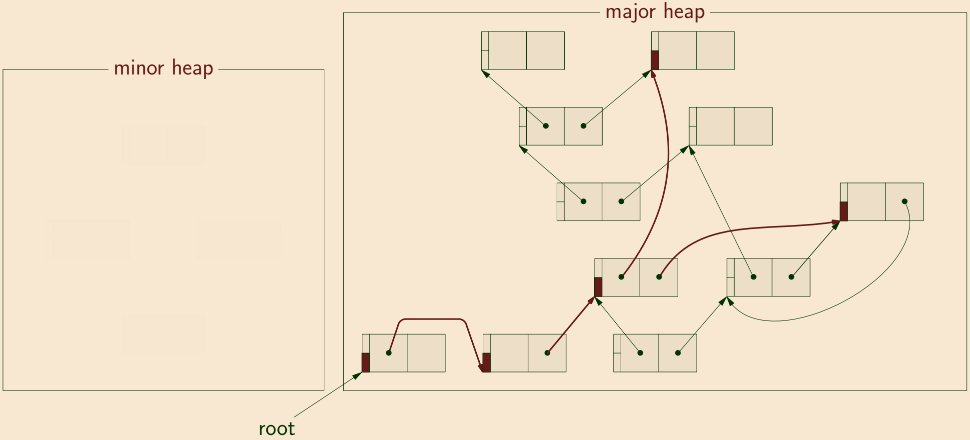
Copying collector for minor heap / mark-and-sweep for major heap



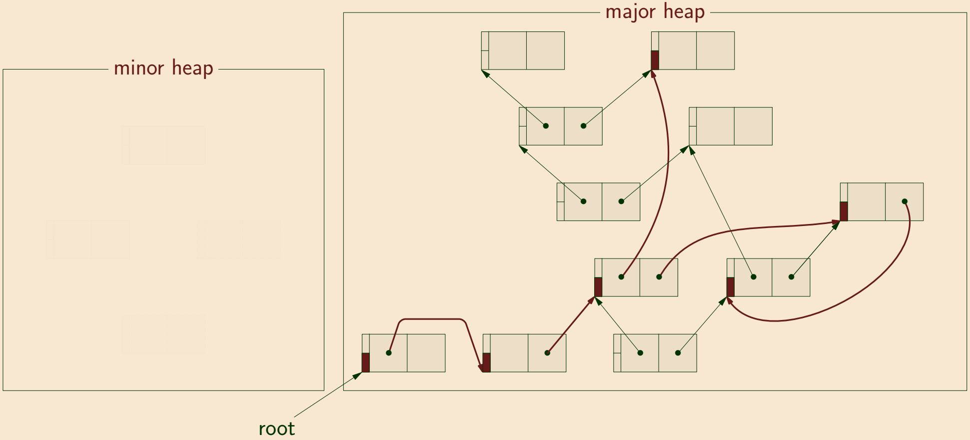
Copying collector for minor heap / mark-and-sweep for major heap



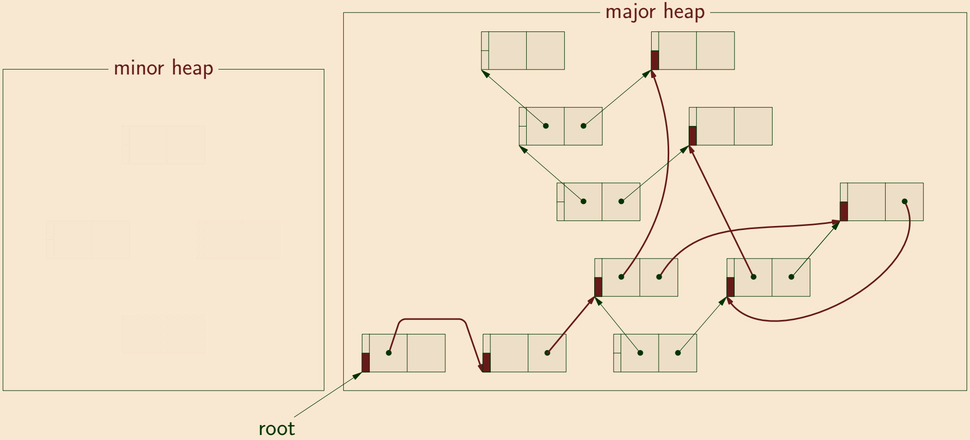
Copying collector for minor heap / mark-and-sweep for major heap



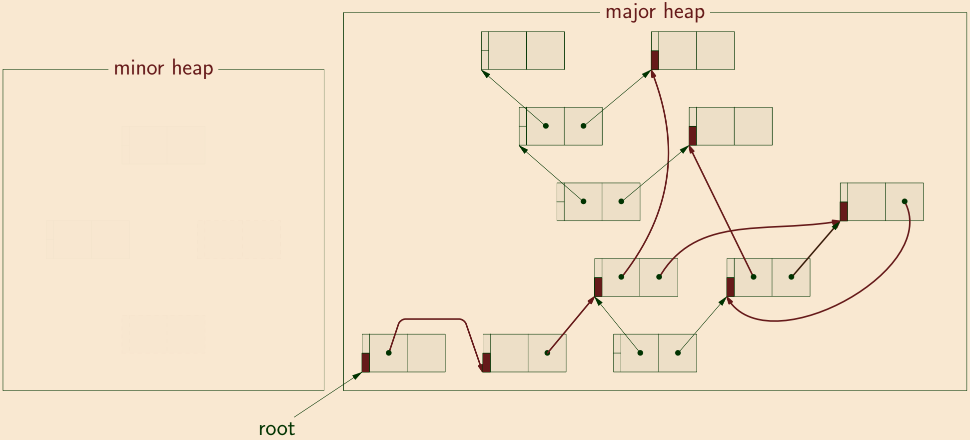
Copying collector for minor heap / mark-and-sweep for major heap



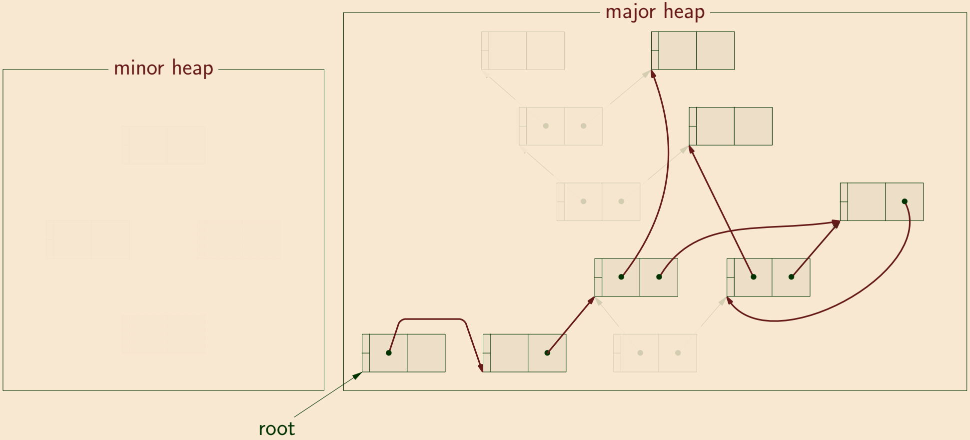
Copying collector for minor heap / mark-and-sweep for major heap



Copying collector for minor heap / mark-and-sweep for major heap



Copying collector for minor heap / mark-and-sweep for major heap



Performance

GC metrics

Throughput: mutator performance

Latency: pauses in mutator execution

Space overhead: e.g. due to mark bits, layout information

More (combination of program behaviour and collector design):

maximum heap size **allocation rate**

collection frequency **mean object size**

proportion of heap occupied by large objects

Performance subtleties

Example

Pause times alone provide little information.
A good **distribution** of pause times is needed for mutators to make progress.

Example

Compaction can slow collection but improve locality (& hence throughput)

Hybrid systems

Many mature systems combine several standard algorithms.

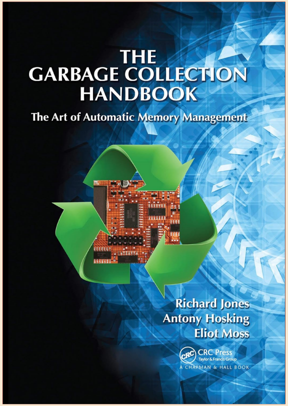
For example, Cedar (1985):

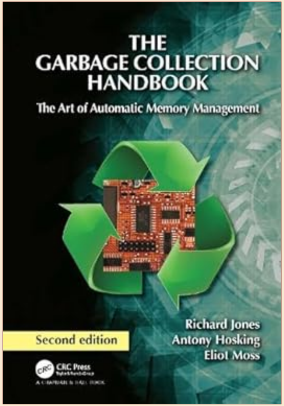
> *"[...] provides both* ***a concurrent reference-counting collector*** *that runs in the background when needed, and* ***a pre-emptive conventional "trace-and-sweep" collector*** *that can be invoked explicitly by the user to reclaim circular data structures [...]*

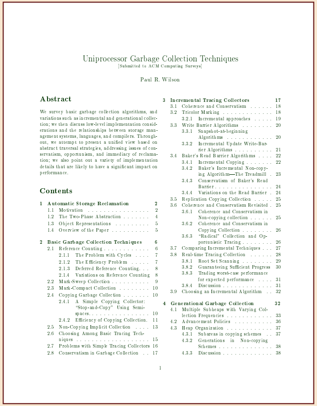
> *"Both collectors treat procedure-call activation records (called frames)* ***"conservatively"****; that is they assume that every ref-sized bit pattern found in a frame might be a ref"*

Reading

Background reading







Uniprocessor Garbage Collection Techniques

[Submitted to ACM Computing Surveys]

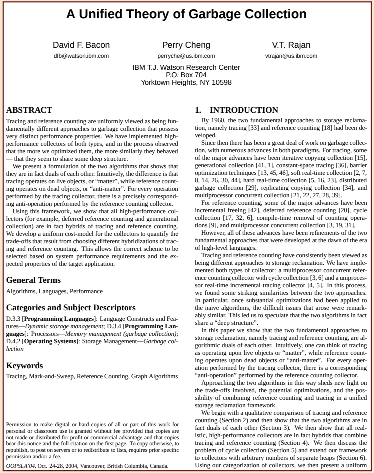
Paul R. Wilson

Abstract

We survey basic garbage collection algorithms, and variations such as incremental and generational collection; we then discuss low-level implementation considerations and the relationships between storage management systems, languages, and compilers. Throughout, we attempt to present a unified view based on abstract traversal strategies, addressing issues of conservatism, opportunism, and immediacy of reclamation; we also point out a variety of implementation details that are likely to have a significant impact on performance.

Contents

1



A Unified Theory of Garbage Collection

David F. Bacon
dfb@watson.ibm.com

Perry Cheng
perryche@us.ibm.com

V.T. Rajan
vtrajan@us.ibm.com

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT

Tracing and reference counting are uniformly viewed as being fundamentally different approaches to garbage collection that possess very distinct performance properties. We have implemented high-performance collectors of both types, and in the process observed that the more we optimized them, the more similarly they behaved — that they seem to share some deep structure.

We present a formulation of the two algorithms that shows that they are in fact duals of each other. Intuitively, the difference is that tracing operates on live objects, or “matter”, while reference counting operates on dead objects, or “anti-matter”. For every operation performed by the tracing collector, there is a precisely corresponding anti-operation performed by the reference counting collector.

Using this framework, we show that all high-performance collectors (for example, deferred reference counting and generational collection) are in fact hybrids of tracing and reference counting. We develop a uniform cost-model for the collectors to quantify the trade-offs that result from choosing different hybridizations of tracing and reference counting. This allows the correct scheme to be selected based on system performance requirements and the expected properties of the target application.

General Terms

Algorithms, Languages, Performance

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*; D.4.2 [**Operating Systems**]: Storage Management—*Garbage collection*

Keywords

Tracing, Mark-and-Sweep, Reference Counting, Graph Algorithms

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

1. INTRODUCTION

By 1960, the two fundamental approaches to storage reclamation, namely tracing [33] and reference counting [18] had been developed.

Since then there has been a great deal of work on garbage collection, with numerous advances in both paradigms. For tracing, some of the major advances have been iterative copying collection [15], generational collection [41, 1], constant-space tracing [36], barrier optimization techniques [13, 45, 46], soft real-time collection [2, 7, 8, 14, 26, 30, 44], hard real-time collection [5, 16, 23], distributed garbage collection [29], replicating copying collection [34], and multiprocessor concurrent collection [21, 22, 27, 28, 39].

For reference counting, some of the major advances have been incremental freeing [42], deferred reference counting [20], cycle collection [17, 32, 6], compile-time removal of counting operations [9], and multiprocessor concurrent collection [3, 19, 31].

However, all of these advances have been refinements of the two fundamental approaches that were developed at the dawn of the era of high-level languages.

Tracing and reference counting have consistently been viewed as being different approaches to storage reclamation. We have implemented both types of collector: a multiprocessor concurrent reference counting collector with cycle collection [3, 6] and a uniprocessor real-time incremental tracing collector [4, 5]. In this process, we found some striking similarities between the two approaches. In particular, once substantial optimizations had been applied to the naïve algorithms, the difficult issues that arose were remarkably similar. This led us to speculate that the two algorithms in fact share a “deep structure”.

In this paper we show that the two fundamental approaches to storage reclamation, namely tracing and reference counting, are algorithmic duals of each other. Intuitively, one can think of tracing as operating upon live objects or “matter”, while reference counting operates upon dead objects or “anti-matter”. For every operation performed by the tracing collector, there is a corresponding “anti-operation” performed by the reference counting collector.

Approaching the two algorithms in this way sheds new light on the trade-offs involved, the potential optimizations, and the possibility of combining reference counting and tracing in a unified storage reclamation framework.

We begin with a qualitative comparison of tracing and reference counting (Section 2) and then show that the two algorithms are in fact duals of each other (Section 3). We then show that all realistic, high-performance collectors are in fact hybrids that combine tracing and reference counting (Section 4). We then discuss the problem of cycle collection (Section 5) and extend our framework to collectors with arbitrary numbers of separate heaps (Section 6). Using our categorization of collectors, we then present a uniform

“**Tracing and reference counting [...] seem to share some deep structure**”

“For every operation performed by the tracing collector, there is a precisely corresponding anti-operation performed by the reference counting collector.”

“[A]ll high-performance collectors [...] are in fact hybrids of tracing and reference counting”

Paper 2: *Hertz and Berger (2005)*

Quantifying the Performance of Garbage Collection vs. Explicit Memory Management

Matthew Hertz *
Computer Science Department
Canisius College
Buffalo, NY 14208
matthew.hertz@canisius.edu

Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
emery@cs.umass.edu

ABSTRACT

Garbage collection yields numerous software engineering benefits, but its quantitative impact on performance remains elusive. One can compare the cost of *conservative* garbage collection to explicit memory management in C/C++ programs by linking in an appropriate collector. This kind of direct comparison is not possible for languages designed for garbage collection (e.g., Java), because programs in these languages naturally do not contain calls to `free`. Thus, the actual gap between the time and space performance of explicit memory management and *precise*, copying garbage collection remains unknown.

We introduce a novel experimental methodology that lets us quantify the performance of precise garbage collection versus explicit memory management. Our system allows us to treat unaltered Java programs as if they used explicit memory management by relying on oracles to insert calls to `free`. These oracles are generated from profile information gathered in earlier application runs. By executing inside an architecturally-detailed simulator, this "oracular" memory manager eliminates the effects of consulting an oracle while measuring the costs of calling `malloc` and `free`. We evaluate two different oracles: a liveness-based oracle that aggressively frees objects immediately after their last use, and a reachability-based oracle that conservatively frees objects just after they are last reachable. These oracles span the range of possible placement of explicit deallocation calls.

We compare explicit memory management to both copying and non-copying garbage collectors across a range of benchmarks using the oracular memory manager, and present real (non-simulated) runs that lend further validity to our results. These results quantify the time-space tradeoff of garbage collection: with five times as much memory, an Appel-style generational collector with a non-copying mature space matches the performance of reachability-based explicit memory management. With only three times as much memory, the collector runs on average 17% slower than explicit memory management. However, with only twice as much memory, garbage collection degrades performance by nearly 70%. When physical memory is scarce, paging causes garbage collection to run an order of magnitude slower than explicit memory management.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Dynamic storage management; D.3.4 [**Processors**]: Memory management (garbage collection)

General Terms

Experimentation, Measurement, Performance

Keywords

oracular memory management, garbage collection, explicit memory management, performance analysis, time-space tradeoff, throughput, paging

1. Introduction

Garbage collection, or automatic memory management, provides significant software engineering benefits over explicit memory management. For example, garbage collection frees programmers from the burden of memory management, eliminates most memory leaks, and improves modularity, while preventing accidental memory overwrites ("dangling pointers") [50, 59]. Because of these advantages, garbage collection has been incorporated as a feature of a number of mainstream programming languages.

Garbage collection can improve programmer productivity [48], but its impact on performance is difficult to quantify. Previous researchers have measured the runtime performance and space impact of *conservative*, non-copying garbage collection in C and C++ programs [19, 62]. For these programs, comparing the performance of explicit memory management to conservative garbage collection is a matter of linking in a library like the Boehm-Demers-Weiser collector [14]. Unfortunately, measuring the performance trade-off in languages designed for garbage collection is not so straightforward. Because programs written in these languages do not explicitly deallocate objects, one cannot simply replace garbage collection with an explicit memory manager. Extrapolating the results of studies with conservative collectors is impossible because precise, relocating garbage collectors (suitable only for garbage-collected languages) consistently outperform conservative, non-relocating garbage collectors [10, 12].

It is possible to measure the costs of garbage collection activity (e.g., tracing and copying) [10, 20, 30, 36, 56] but it is impossible to subtract garbage collection's effect on mutator performance. Garbage collection alters application behavior both by visiting and reorganizing memory. It also degrades locality, especially when physical memory is scarce [61]. Subtracting the costs of garbage collection also ignores the improved locality that explicit memory managers can provide by immediately recycling just-freed memory [53, 55, 57, 58]. For all these reasons, the costs of precise,

*Work performed at the University of Massachusetts Amherst.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

"[A] novel experimental methodology that lets us quantify the performance of **precise garbage collection versus explicit memory management**."

"[W]ith five times as much memory, an Appel-style generational collector with a non-copying mature space matches the performance of reachability-based explicit memory management."

"When physical memory is scarce, paging causes garbage collection to run an order of magnitude slower than explicit memory management"

Paper 3: *Shahriyar et al (2014)*

Algorithms

Performance

Reading

Fast Conservative Garbage Collection

Rifat Shahriyar
Australian National University
Rifat.Shahriyar@anu.edu.au

Stephen M. Blackburn
Australian National University
Steve.Blackburn@anu.edu.au

Kathryn S. McKinley
Microsoft Research
mckinley@microsoft.com

Abstract

Garbage collectors are exact or conservative. An *exact* collector identifies all references precisely and may move referents and update references, whereas a *conservative* collector treats one or more of stack, register, and heap references as ambiguous. *Ambiguous references* constrain collectors in two ways. (1) Since they may be pointers, the collectors must retain referents. (2) Since they may be values, the collectors cannot modify them, pinning their referents.

We explore *conservative* collectors for managed languages, with ambiguous stacks and registers. We show that for Java benchmarks they retain and pin remarkably few heap objects: <0.01% are falsely retained and 0.03% are pinned. The larger effect is collector design. Prior conservative collectors (1) use mark-sweep and unnecessarily forgo moving all objects, or (2) use mostly copying and pin entire pages. Compared to generational collection, overheads are substantial: 12% and 45% respectively. We introduce high performance conservative Immix and reference counting (RC). Immix is a mark-region collector with fine *line*-grain pinning and *opportunistic* copying of unambiguous referents. Deferred RC simply needs an object map to deliver the first conservative RC. We implement six exact collectors and their conservative counterparts. Conservative Immix and RC come within 2 to 3% of their exact counterparts. In particular, conservative RC Immix is slightly *faster* than a well-tuned exact generational collector. These findings show that for managed languages, conservative collection is compatible with high performance.

Categories and Subject Descriptors Software, Virtual Machines, Memory management, Garbage collection

Keywords Conservative, Reference Counting, Immix, Mark-Region

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10…$15.00.
http://dx.doi.org/10.1145/2660193.2660198

1. Introduction

Language semantics and compiler implementations determine whether memory managers may implement *exact* or *conservative* garbage collection. Exact collectors identify all references and may move objects and redirect references transparently to applications. Conservative collectors must reason about *ambiguous references*, constraining them in two ways. (1) Because ambiguous references may be pointers, *the collector must conservatively retain referents*. (2) Because ambiguous references may be values, *the collector must not change them* and cannot move (must pin) the referent.

Languages such as C and C++ are not memory safe: programs may store and manipulate pointers directly. Consequently, their compilers cannot prove whether any value is a pointer or not, which forces their collectors to be conservative and non-moving. Managed languages, such as Java, C#, Python, PHP, JavaScript, and safe C variants, have a choice between exact and conservative collection. In principle, a conservative collector for managed languages may treat stacks, registers, heap, and other references conservatively. In practice, the type system easily identifies heap references exactly. However, many systems for JavaScript, PHP, Objective C, and other languages treat ambiguous references in stacks and registers conservatively.

This paper explores conservative collectors with ambiguous stacks and registers. We first show that the direct consequences of these ambiguous references on *excess retention* and *pinning* are surprisingly low. Using a Java Virtual Machine and 18 Java benchmarks, conservative roots falsely retain less than 0.01% of objects and pin less than 0.03%. However, conservative constraints have had a large indirect cost by how they shaped garbage collection algorithms.

Many widely used managed systems implement collectors that are conservative with respect to stacks and registers. Microsoft's Chakra JavaScript VM implements a conservative mark-sweep Boehm, Demers, Weiser style (BDW) collector [15, 19]. This non-moving free-list collector was originally proposed for C, but some managed runtimes use it directly and many others have adapted it. Apple's WebKit JavaScript VM implements a Mostly Copying Conservative (MCC) collector, also called a Bartlett-style collec-

"Garbage collectors are **exact or conservative**. […] We explore *conservative* collectors for managed languages, with ambiguous stacks and registers. We show that for Java benchmarks they retain and pin remarkably few heap objects"

"We introduce high performance conservative Immix and reference counting (RC)."

"[F]or managed languages, conservative collection is compatible with high performance."