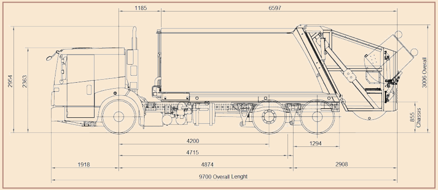
Garbage collection



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)

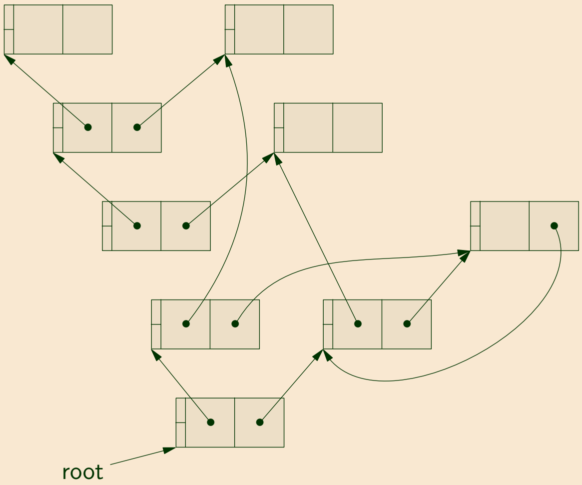
An object is **live** if a mutator will access it in the future

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

Mark-and-sweep collection

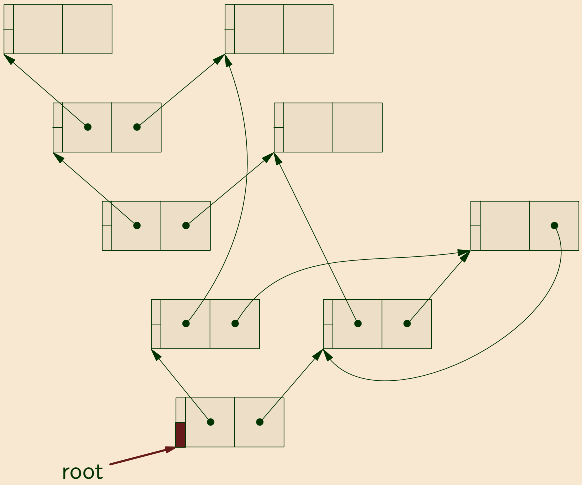
Mark

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



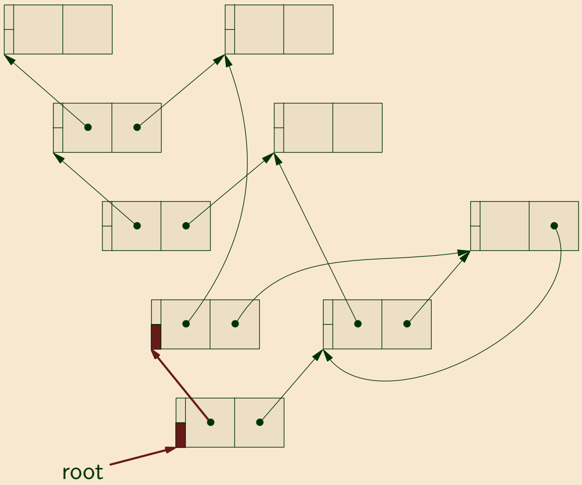
Mark

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



Mark

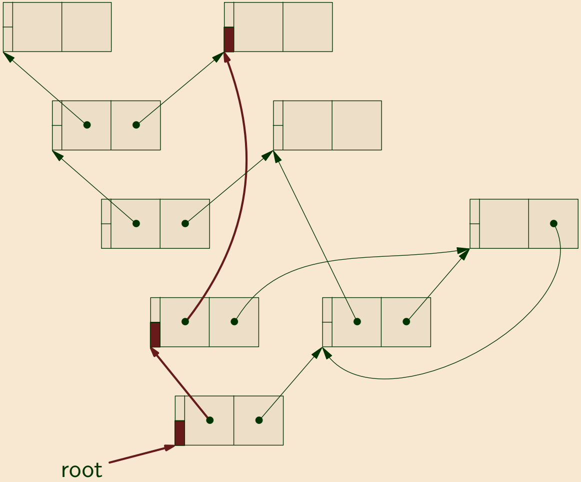
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark-and-sweep collection

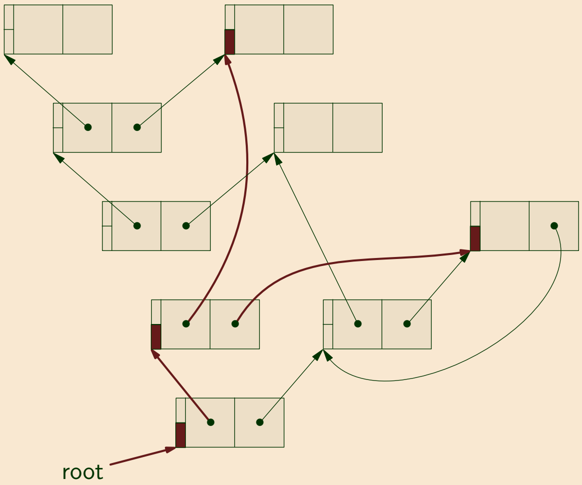
Mark

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



Mark

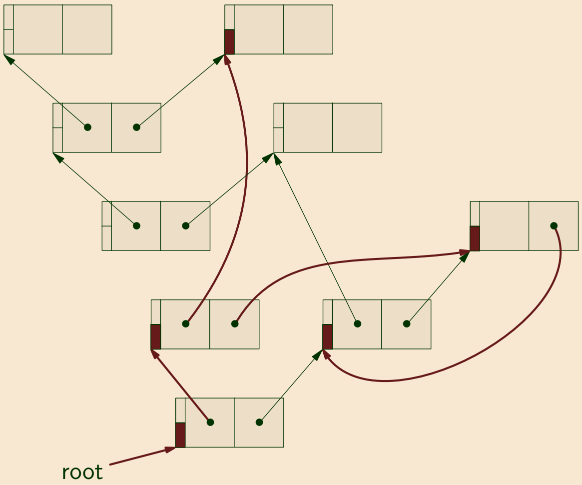
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark-and-sweep collection

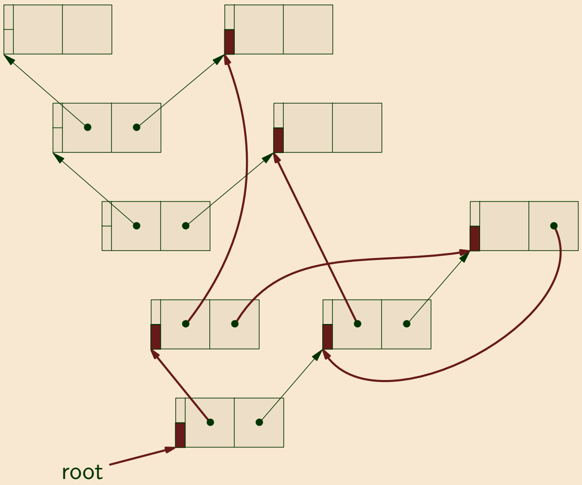
Mark

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



Mark

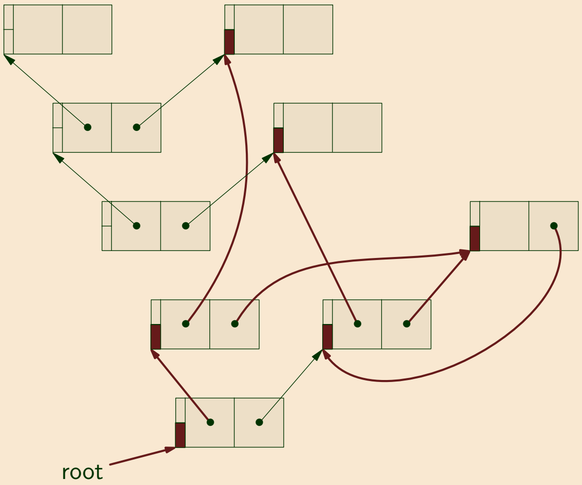
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark-and-sweep collection

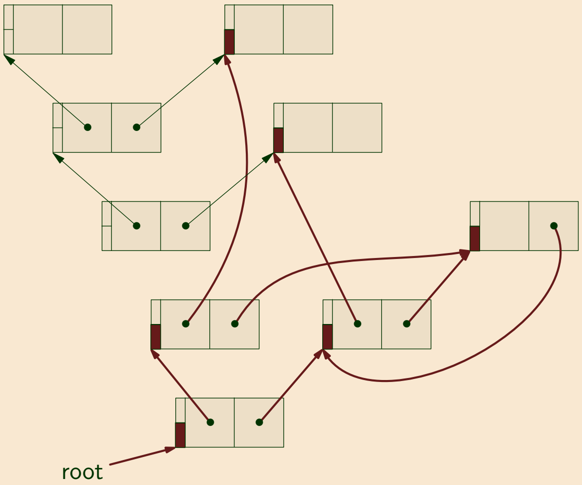
Mark

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



Mark

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



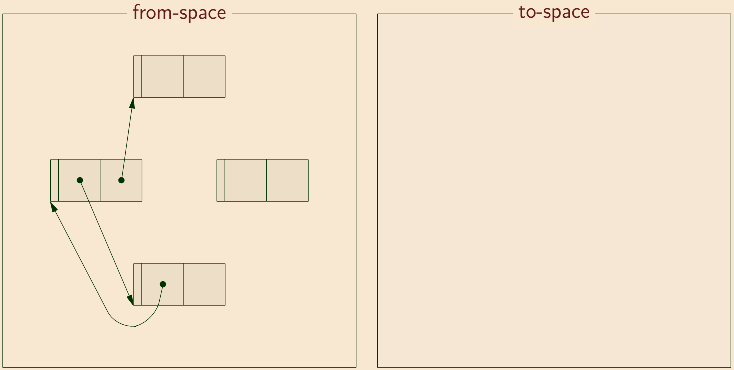
Copying collection

Collect

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

leave **forwarding addresses** in from-space

switch roles of spaces



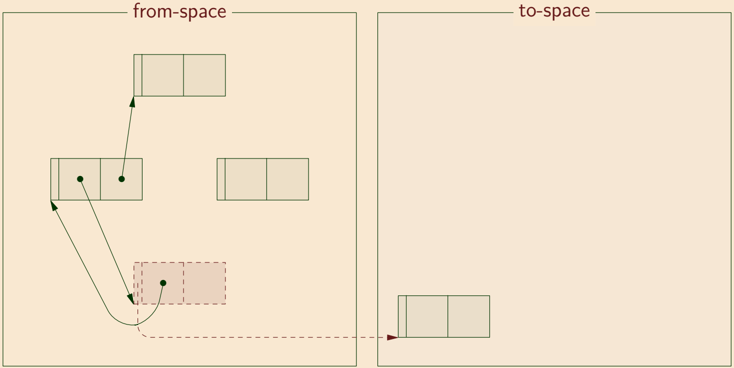
Copying collection

Collect

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

leave **forwarding addresses** in from-space

switch roles of spaces



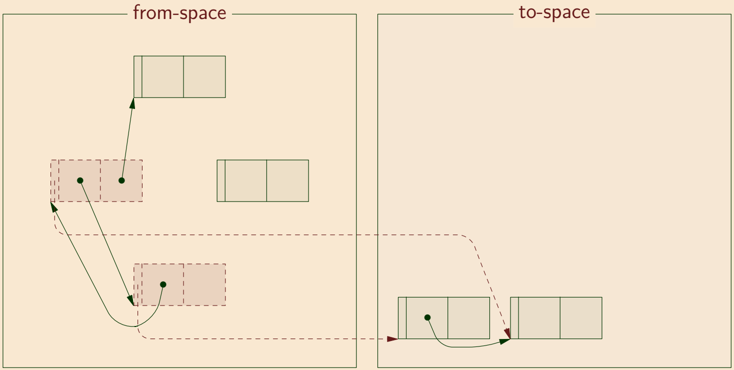
Copying collection

Collect

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

leave **forwarding addresses** in from-space

switch roles of spaces



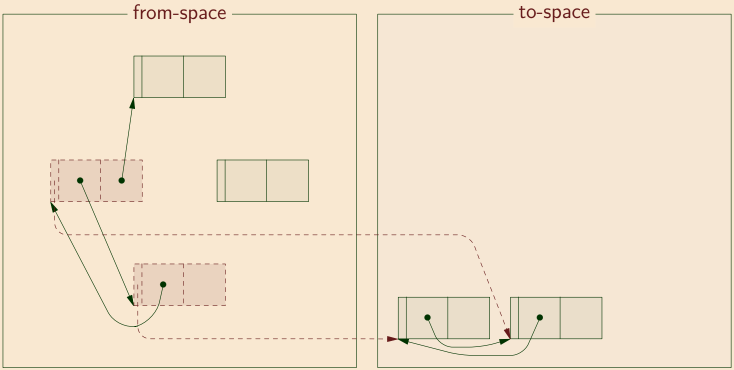
Copying collection

Collect

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

leave **forwarding addresses** in from-space

switch roles of spaces



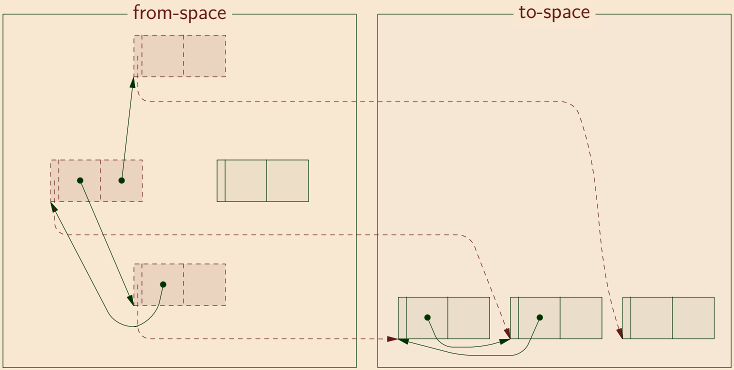
Copying collection

Collect

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

leave **forwarding addresses** in from-space

switch roles of spaces



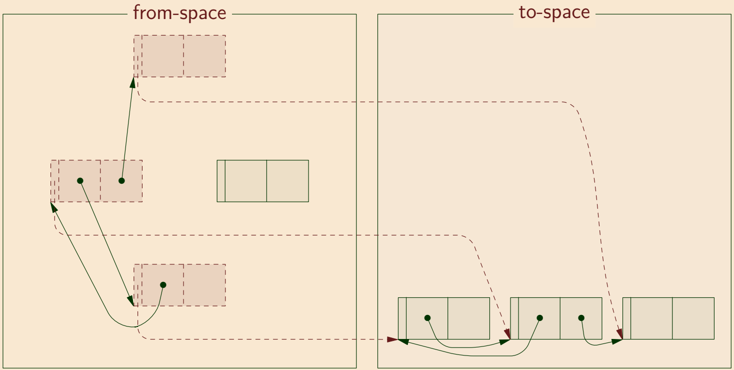
Copying collection

Collect

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

leave **forwarding addresses** in from-space

switch roles of spaces



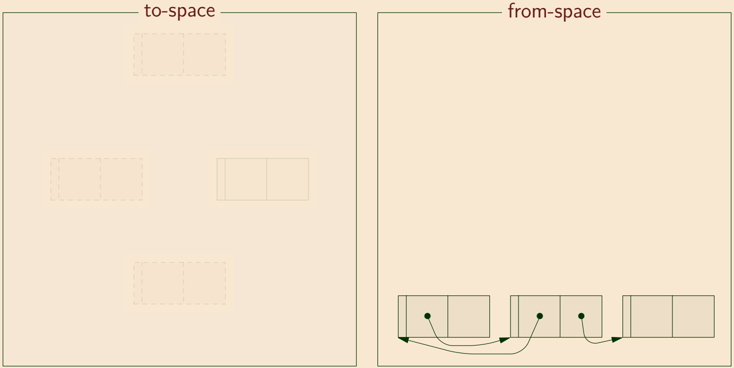
Copying collection

Collect

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

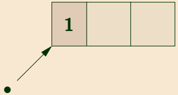
leave **forwarding addresses** in from-space

switch roles of spaces

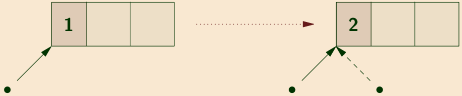


The **reference count** tracks the number of pointers to each object.

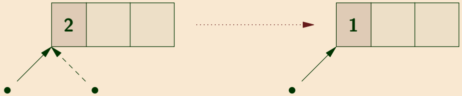
An object's reference count is 1 when the object is created:



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)

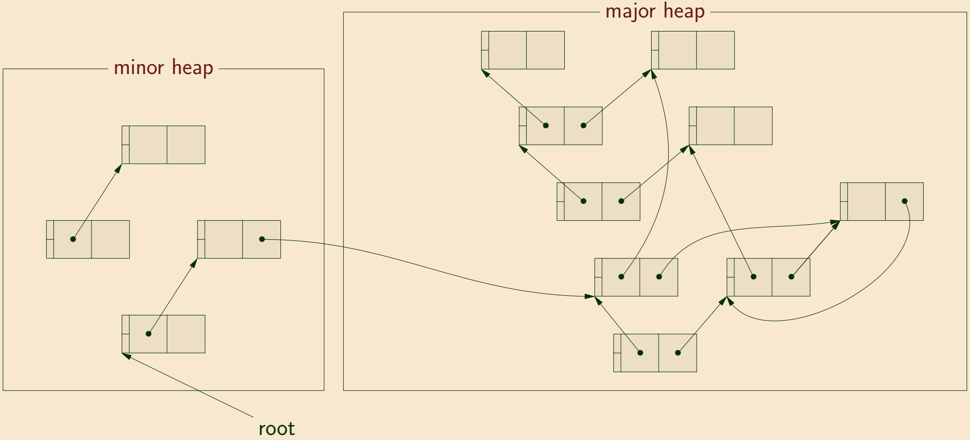
Idea: use an approximation to guess whether a value represents a pointer, e.g.:

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

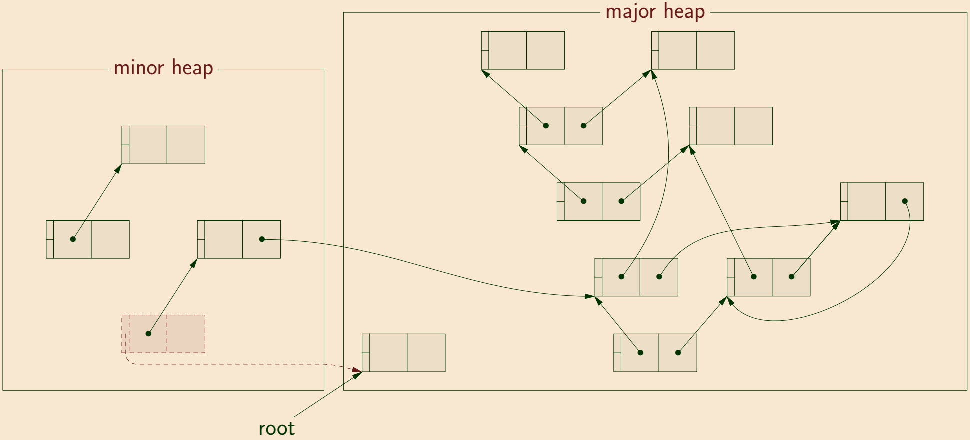
Drawbacks

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

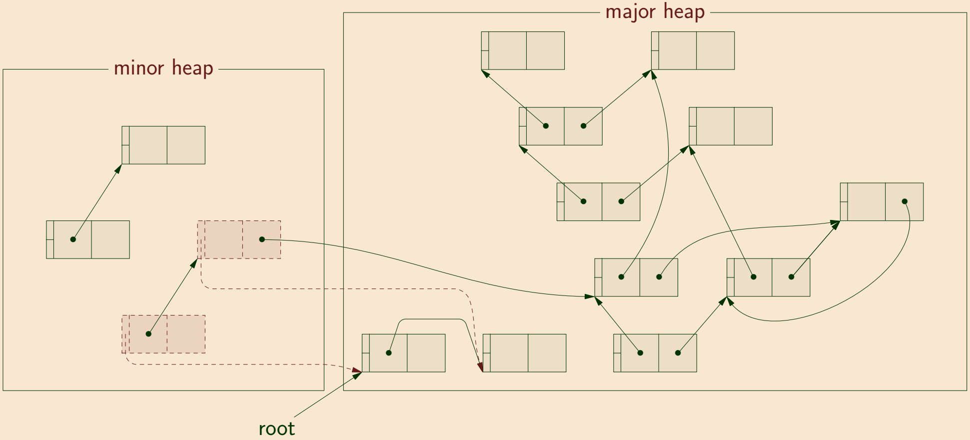
Copying collector for minor heap / mark-and-sweep for major heap



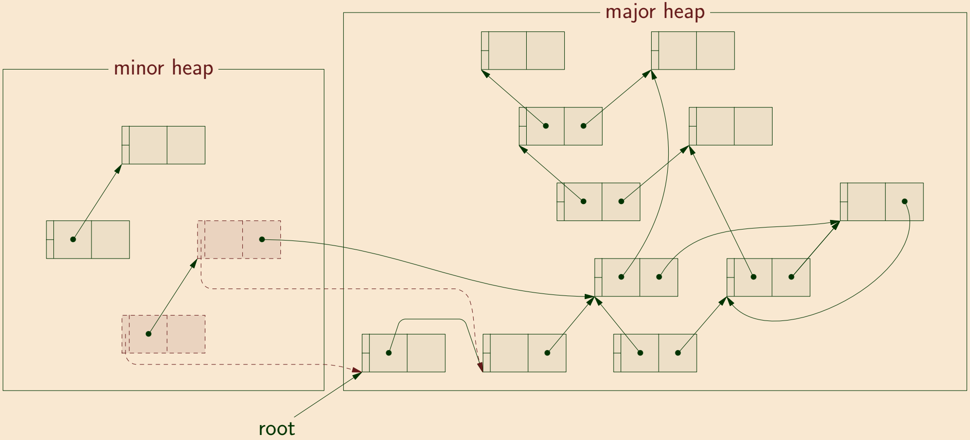
Copying collector for minor heap / mark-and-sweep for major heap



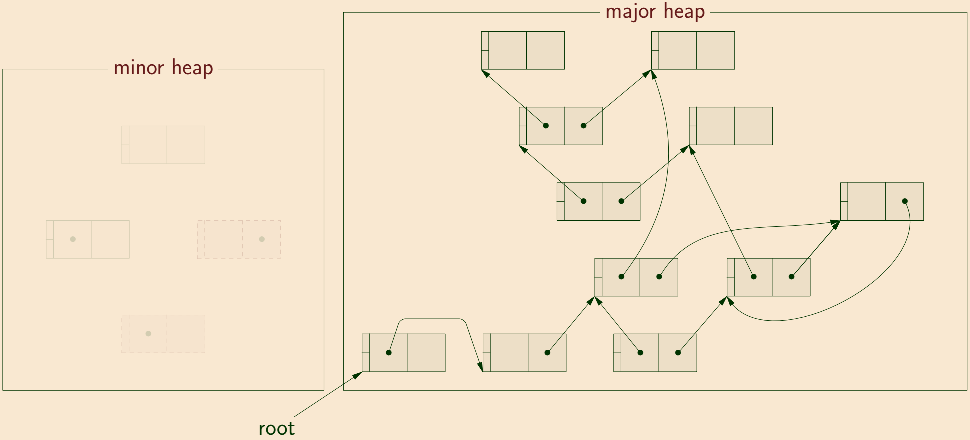
Copying collector for minor heap / mark-and-sweep for major heap



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

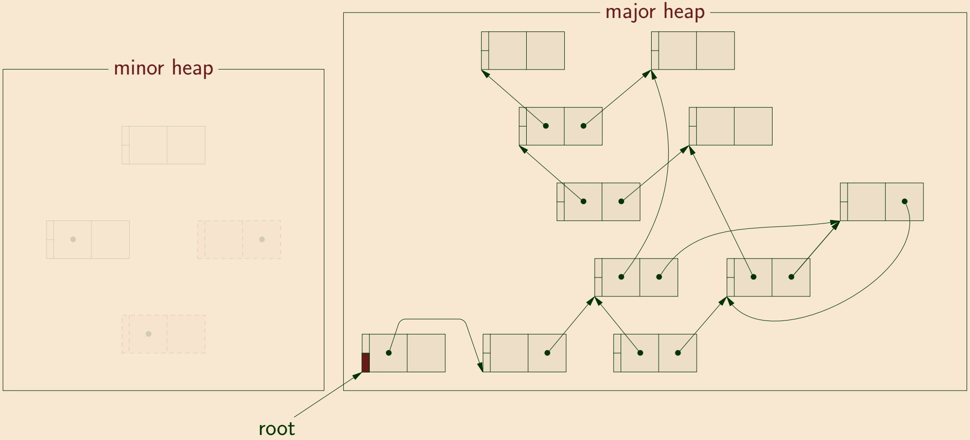


Copying collector for minor heap / mark-and-sweep for major heap

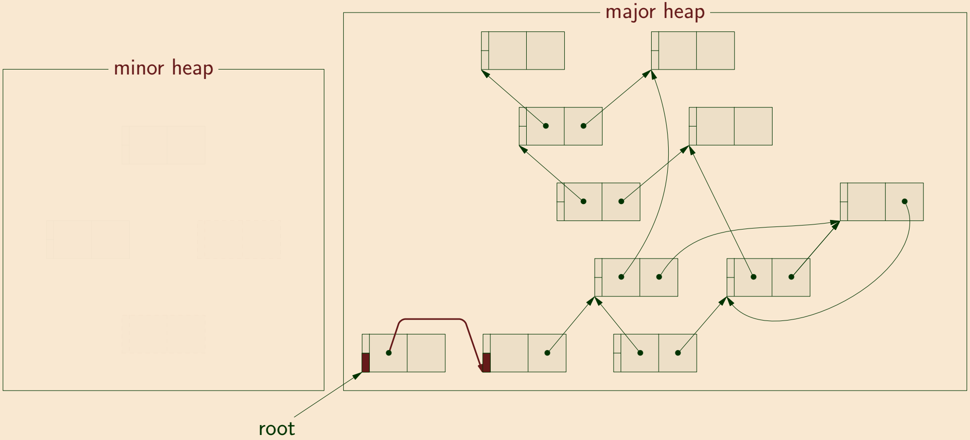


Generational collection

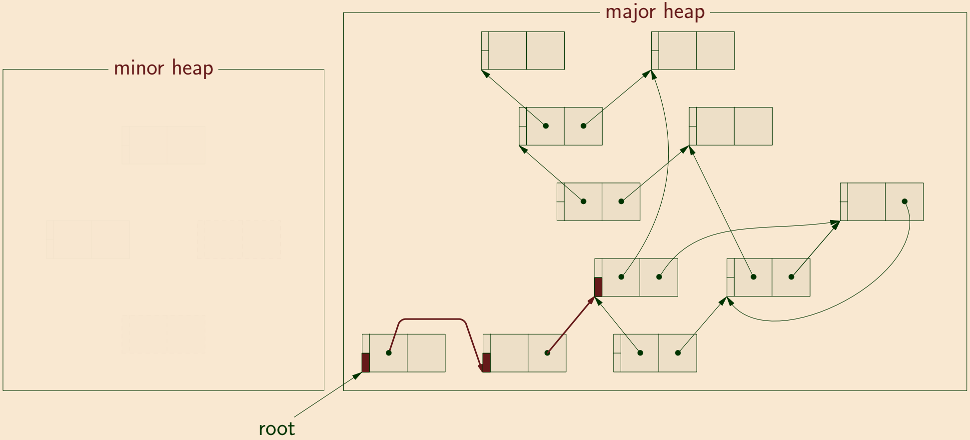
Copying collector for minor heap / mark-and-sweep for major heap



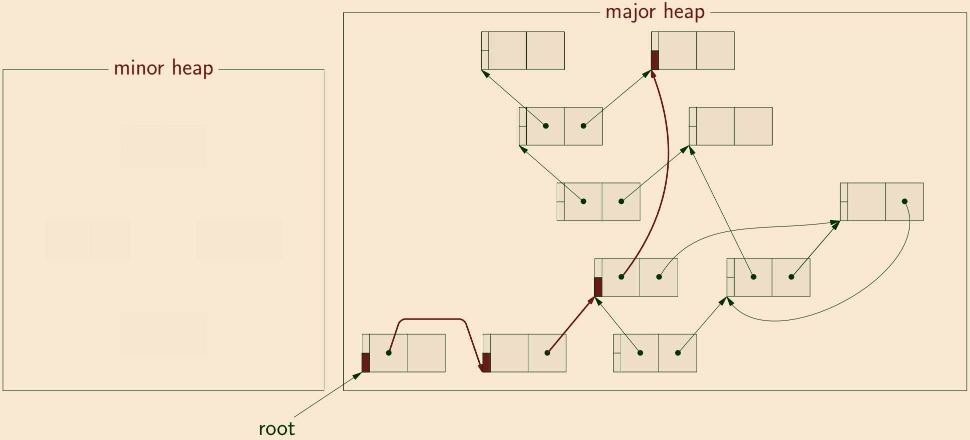
Copying collector for minor heap / mark-and-sweep for major heap



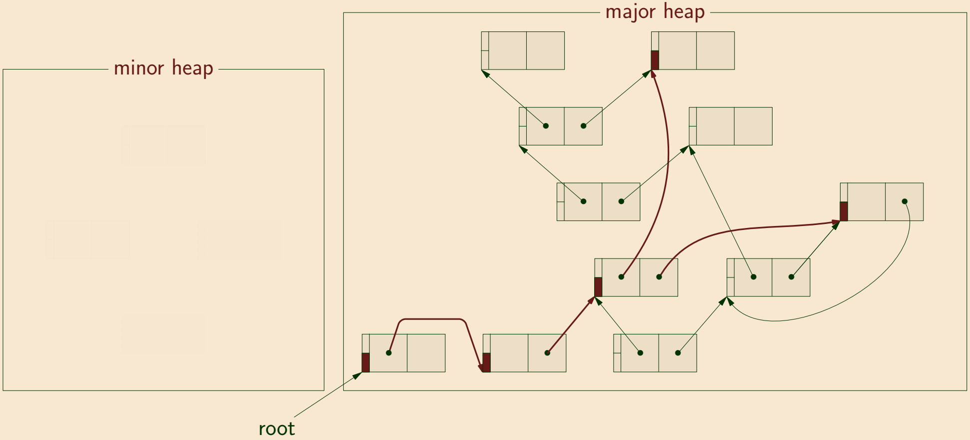
Copying collector for minor heap / mark-and-sweep for major heap



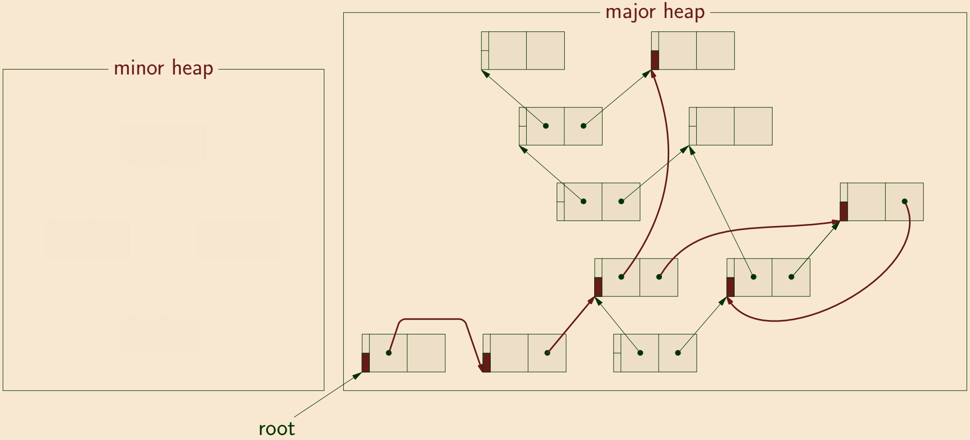
Copying collector for minor heap / mark-and-sweep for major heap



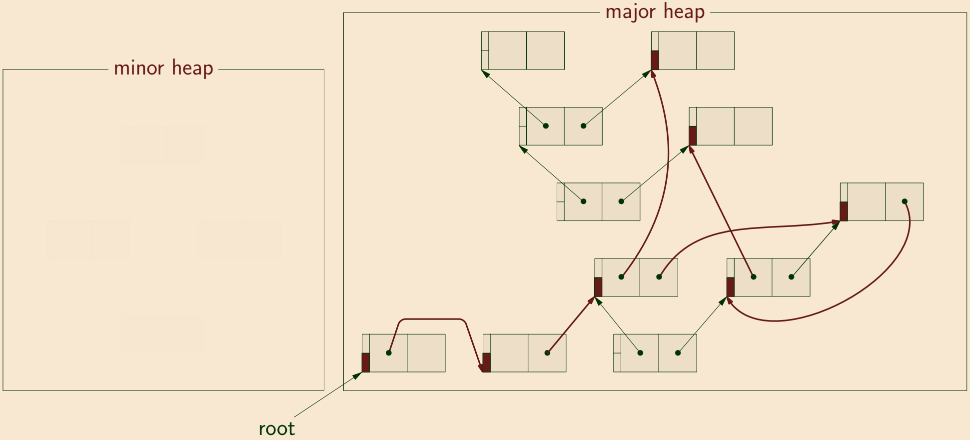
Copying collector for minor heap / mark-and-sweep for major heap



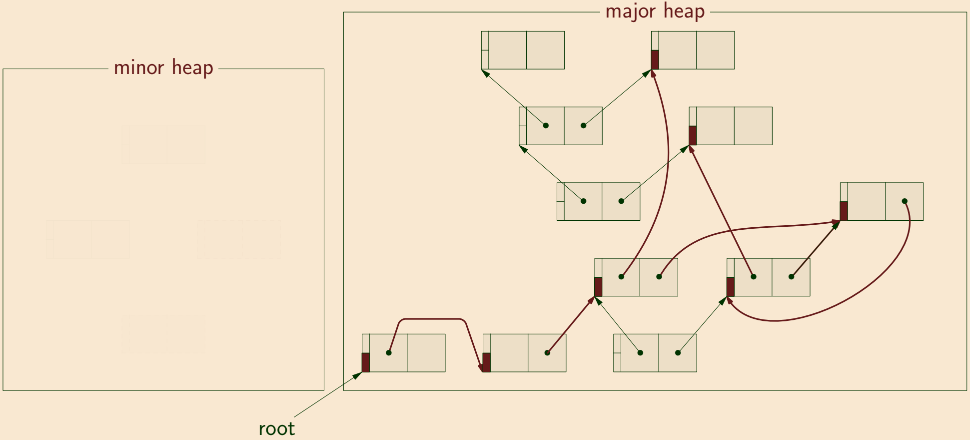
Copying collector for minor heap / mark-and-sweep for major heap



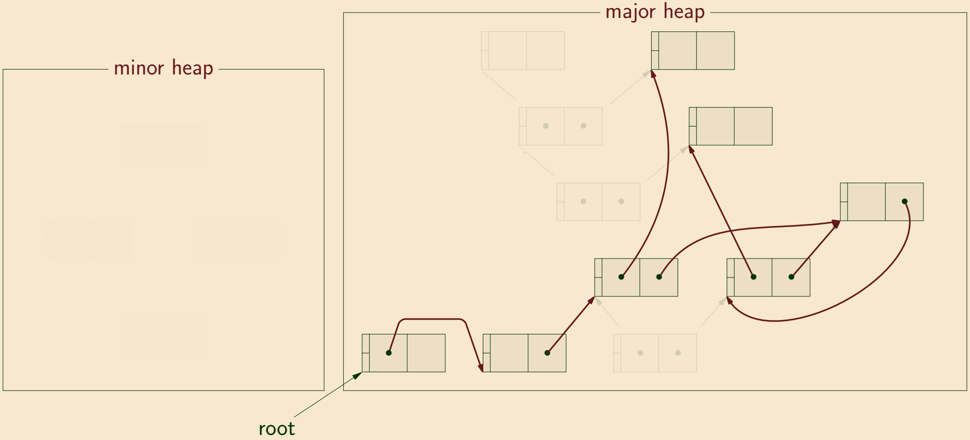
Copying collector for minor heap / mark-and-sweep for major heap



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

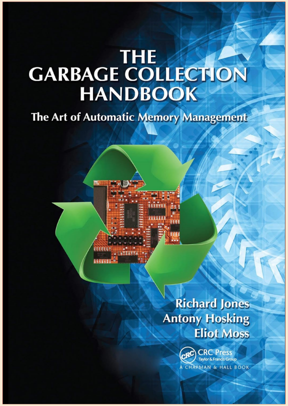
Many mature systems combine several standard algorithms.

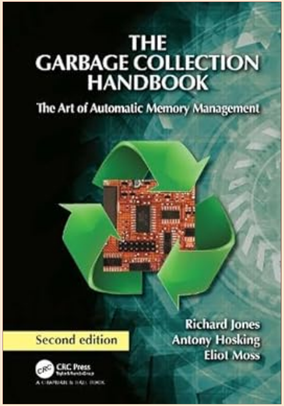
For example, Cedar (1985):

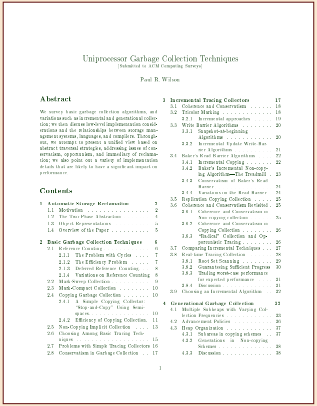
> *"[...] provides both* ***a concurrent reference-counting collector*** *that runs in the background when needed, and* ***a pre-emptive conventional "trace-and-sweep" collector*** *that can be invoked explicitly by the user to reclaim circular data structures [...]*
>
> *"Both collectors treat procedure-call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

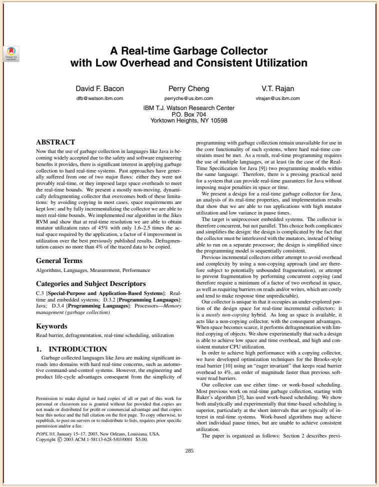
Abstract

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

Paper 1: *Bacon et al (2003)*



A Real-time Garbage Collector with Low Overhead and Consistent Utilization

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

Now that the use of garbage collection in languages like Java is becoming widely accepted due to the safety and software engineering benefits it provides, there is significant interest in applying garbage collection to hard real-time systems. Past approaches have generally suffered from one of two major flaws: either they were not provably real-time, or they imposed large space overheads to meet the real-time bounds. We present a mostly non-moving, dynamically defragmenting collector that overcomes both of these limitations: by avoiding copying in most cases, space requirements are kept low; and by fully incrementalizing the collector we are able to meet real-time bounds. We implemented our algorithm in the Jikes RVM and show that at real-time resolution we are able to obtain mutator utilization rates of 45% with only 1.6–2.5 times the actual space required by the application, a factor of 4 improvement in utilization over the best previously published results. Defragmentation causes no more than 4% of the traced data to be copied.

General Terms

Algorithms, Languages, Measurement, Performance

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Java; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

Keywords

Read barrier, defragmentation, real-time scheduling, utilization

1. INTRODUCTION

Garbage collected languages like Java are making significant inroads into domains with hard real-time concerns, such as automotive command-and-control systems. However, the engineering and product life-cycle advantages consequent from the simplicity of programming with garbage collection remain unavailable for use in the core functionality of such systems, where hard real-time constraints must be met. As a result, real-time programming requires the use of multiple languages, or at least (in the case of the Real-Time Specification for Java [9]) two programming models within the same language. Therefore, there is a pressing practical need for a system that can provide real-time guarantees for Java without imposing major penalties in space or time.

We present a design for a real-time garbage collector for Java, an analysis of its real-time properties, and implementation results that show that we are able to run applications with high mutator utilization and low variance in pause times.

The target is uniprocessor embedded systems. The collector is therefore concurrent, but not parallel. This choice both complicates and simplifies the design: the design is complicated by the fact that the collector must be interleaved with the mutators, instead of being able to run on a separate processor; the design is simplified since the programming model is sequentially consistent.

Previous incremental collectors either attempt to avoid overhead and complexity by using a non-copying approach (and are therefore subject to potentially unbounded fragmentation), or attempt to prevent fragmentation by performing concurrent copying (and therefore require a minimum of a factor of two overhead in space, as well as requiring barriers on reads and/or writes, which are costly and tend to make response time unpredictable).

Our collector is unique in that it occupies an under-explored portion of the design space for real-time incremental collectors: it is a *mostly non-copying* hybrid. As long as space is available, it acts like a non-copying collector, with the consequent advantages. When space becomes scarce, it performs defragmentation with limited copying of objects. We show experimentally that such a design is able to achieve low space and time overhead, and high and consistent mutator CPU utilization.

In order to achieve high performance with a copying collector, we have developed optimization techniques for the Brooks-style read barrier [10] using an "eager invariant" that keeps read barrier overhead to 4%, an order of magnitude faster than previous software read barriers.

Our collector can use either time- or work-based scheduling. Most previous work on real-time garbage collection, starting with Baker's algorithm [5], has used work-based scheduling. We show both analytically and experimentally that time-based scheduling is superior, particularly at the short intervals that are typically of interest in real-time systems. Work-based algorithms may achieve short individual pause times, but are unable to achieve consistent utilization.

The paper is organized as follows: Section 2 describes previ-

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.

"[...] there is significant interest in applying garbage collection to hard real-time systems."

"Past approaches have generally suffered from one of two major flaws: either they were not provably real-time, or they imposed large space overheads to meet the real-time bounds."

"We [...] show that at real-time resolution we are able to obtain mutator utilization rates of 45% with only 1.6–2.5 times the actual space required by the application"

Concurrent GCs and Modern Java Workloads: A Cache Perspective

Maria Carpen-Amarie
Huawei Zurich Research Center
maria.carpen.amarie@huawei.com

Georgios Vavouliotis
Huawei Zurich Research Center
georgios.vavouliotis2@huawei.com

Konstantinos Tovletoglou
Huawei Zurich Research Center
konstantinos.tovletoglou@huawei.com

Boris Grot
University of Edinburgh
Huawei Zurich Research Center
boris.grot@ed.ac.uk

Rene Mueller
Huawei Zurich Research Center
rene.mueller@huawei.com

Abstract

The garbage collector (GC) is a crucial component of language runtimes, offering correctness guarantees and high productivity in exchange for a run-time overhead. Concurrent collectors run alongside application threads (mutators) and share CPU resources. A likely point of contention between mutators and GC threads and, consequently, a potential overhead source is the shared last-level cache (LLC).

This work builds on the hypothesis that the cache pollution caused by concurrent GCs hurts application performance. We validate this hypothesis with a cache-sensitive Java micro-benchmark. We find that concurrent GC activity may slow down the application by up to $3\times$ and increase the LLC misses by 3 orders of magnitude. However, when we extend our analysis to a suite of benchmarks representative for today's server workloads (Renaissance), we find that only 5 out of 23 benchmarks show a statistically significant correlation between GC-induced cache pollution and performance. Even for these, the performance overhead of GC does not exceed 10 %. Based on further analysis, we conclude that the lower impact of the GC on the performance of Renaissance benchmarks is due to their lack of sensitivity to LLC capacity.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; **Garbage collection**; • **Computer systems organization** → **Multicore architectures**.

Keywords: JVM, garbage collection, ZGC, cache pollution

ISMM '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0179-5/23/06…$15.00
https://doi.org/10.1145/3591195.3595269

ACM Reference Format:
Maria Carpen-Amarie, Georgios Vavouliotis, Konstantinos Tovletoglou, Boris Grot, and Rene Mueller. 2023. Concurrent GCs and Modern Java Workloads: A Cache Perspective. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3591195.3595269

1 Introduction

Automatic memory management, also known as *garbage collection (GC)*,[1] is a technique that provides memory access safety and reliability while significantly reducing developers' load. These aspects make garbage collection an essential component of managed runtime environments (*e.g.*, Java, C#, JavaScript), which are intensively used by web services (*e.g.*, Twitter), web browsers, and mobile platforms (*e.g.*, Android). For these reasons, automatic memory management continues to be a hot topic today, even after more than half a century of active research towards its optimization.

However, the benefits offered by the GC do not come for free. Prior work [2, 4, 13, 19, 27, 30] has shown that application performance is profoundly contingent on the effectiveness of the garbage collector. Historically, GCs harmed application performance due to unacceptably long stop-the-world (STW) pauses. To reduce the performance overhead and responsiveness issues created by STW events, significant effort was put into re-designing and implementing more efficient GCs [7, 12, 16, 22, 26, 28, 29]. Therefore, most modern GCs [10, 32] have increasingly shorter pauses, while performing most of the work concurrently with the application threads (*e.g.*, ZGC [32]). However, concurrency comes at a price as well [5], as the application needs to share resources with the GC (*e.g.*, cache capacity, bandwidth, CPU time) and even sometimes help with the collection itself. To the best of our knowledge, there is no work on concurrent GCs that quantifies these overhead components individually. Such information would reveal new weaknesses and opportunities and facilitate well-targeted performance improvements.

[1] For the rest of the paper we use *GC* to refer to the process of garbage collection as well as the garbage collector itself.

"This work builds on the hypothesis that the cache pollution caused by concurrent GCs hurts application performance."

"We find that concurrent GC activity may slow down the application by up to $3\times$ and increase the LLC misses by 3 orders of magnitude."

"However, [...] we find that only 5 out of 23 benchmarks show a statistically significant correlation between GC-induced cache pollution and performance."

Paper 3: *Zhao et al (2022)*

Low-Latency, High-Throughput Garbage Collection

Wenyu Zhao
School of Computing
Australian National University
Australia
wenyu.zhao@anu.edu.au

Stephen M. Blackburn
School of Computing
Australian National University
Australia
steve.blackburn@anu.edu.au

Kathryn S. McKinley
Google
United States
ksmckinley@google.com

Abstract

To achieve short pauses, state-of-the-art concurrent copying collectors such as C4, Shenandoah, and ZGC use substantially more CPU cycles and memory than simpler collectors. They suffer from design limitations: i) concurrent copying with inherently expensive read and write barriers, ii) scalability limitations due to tracing, and iii) immediacy limitations for mature objects that impose memory overheads.

This paper takes a different approach to optimizing responsiveness and throughput. It uses the insight that regular, brief stop-the-world collections deliver sufficient responsiveness at greater efficiency than concurrent evacuation. It introduces LXR, where stop-the-world collections use reference counting (RC) and judicious copying. RC delivers scalability and immediacy, promptly reclaiming young and mature objects. RC, in a hierarchical Immix heap structure, reclaims most memory without any copying. Occasional concurrent tracing identifies cyclic garbage. LXR introduces: i) RC remembered sets for judicious copying of mature objects; ii) a novel low-overhead write barrier that combines coalescing reference counting, concurrent tracing, and remembered set maintenance; iii) object reclamation while performing a concurrent trace; iv) lazy processing of decrements; and v) novel survival rate triggers that modulate pause durations.

LXR combines excellent responsiveness and throughput, improving over production collectors. On the widely-used Lucene search engine in a tight heap, LXR delivers 7.8× better throughput and 10× better 99.99% tail latency than Shenandoah. On 17 diverse modern workloads in a moderate heap, LXR outperforms OpenJDK's default G1 on throughput by 4% and Shenandoah by 43%.

CCS Concepts: **• Software and its engineering → Garbage collection**;

Keywords: Garbage collection, Reference counting

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523440

ACM Reference Format:
Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-Latency, High-Throughput Garbage Collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523440

1 Introduction

Modern concurrent garbage collectors are surprisingly expensive. Growth in memory footprints and latency-sensitive applications led vendors to focus on low pause time collectors, such as C4 [39], Shenandoah [20], and ZGC [27]. While they achieve low pause times, they do so at memory and CPU costs. Furthermore, we show that their low pause times do not always translate into low latency for latency-sensitive applications. This paper identifies why they are expensive and proposes a very different approach. We introduce LXR (**L**atency-critical Immi**X** with **R**eference counting), implement it in OpenJDK, and compare it against these widely-used collectors on diverse contemporary workloads.

The early Garbage First (G1) collector is a copying collector with concurrent tracing [17]. Each collection chooses a set of profitable fixed size regions to evacuate. It is optimized for throughput and low pause times. C4, Shenandoah, and ZGC built on the G1 foundation, seeking to further reduce pause times, believing lower pause times would translate to improved application latency. These collectors all rely on i) concurrent tracing to identify live objects, ii) evacuation alone to reclaim space, and iii) expensive read and/or write barriers. These choices have fundamental implications on i) application (mutator) performance, ii) timeliness of reclamation, and iii) collector performance. They reclaim memory only with concurrent copying, an intrinsically expensive approach that relocates every object using expensive barriers to maintain coherence of heap references [34].

Table 1 shows the tradeoff Shenandoah makes to achieve low pause times and that those low pause times *do not* translate to low latency on the widely used Lucene enterprise search engine (lusearch). It compares G1—the OpenJDK default, optimized for throughput; Shenandoah–optimized for latency; and LXR—optimized for both, using a tight heap 1.3× the minimum required by G1 on our Zen 3. The workload is challenging because it is highly parallel and has a very high allocation rate. We report *throughput* using queries per second (QPS) and total time, and *latency* using query

"To achieve short pauses, state-of-the-art concurrent copying collectors such as C4, Shenandoah, and ZGC use substantially more CPU cycles and memory than simpler collectors."

"This paper [...] uses the insight that regular, brief stop-the-world collections deliver sufficient responsiveness at greater efficiency than concurrent evacuation."

"[...] LXR delivers $7.8\times$ better throughput and $10\times$ better 99.99% tail latency than Shenandoah."