University of Glasgow

Garbage Collection

Advanced Systems Programming (M)

Lecture 5

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866,

Rationale

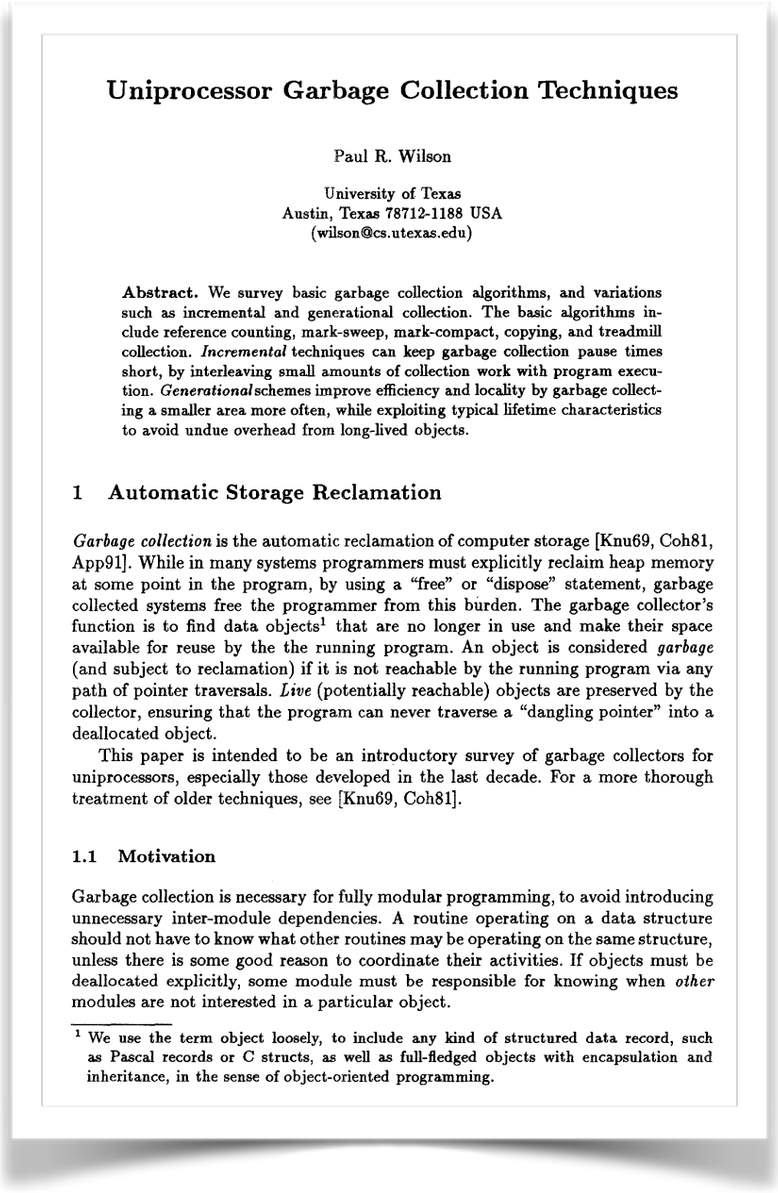
- Region-based memory management (→ lecture 4) is novel, trades program complexity for predictable resource management
- Garbage collection widely implemented, but less predictable

- Need to understand garbage collector operation to understand the performance-complexity trade-off

cc BY ND

Lecture Outline

- Garbage collection algorithms
  - Mark-sweep
  - Mark-compact
  - Copying collectors
  - Generational collectors
  - Incremental collectors
- Real-time garbage collection
- Practical factors



Uniprocessor Garbage Collection Techniques

Paul R. Wilson

University of Texas
Austin, Texas 78712-1188 USA
(wilson@cs.utexas.edu)

Abstract. We survey basic garbage collection algorithms, and variations such as incremental and generational collection. The basic algorithms include reference counting, mark-sweep, mark-compact, copying, and treadmill collection. *Incremental* techniques can keep garbage collection pause times short, by interleaving small amounts of collection work with program execution. *Generational* schemes improve efficiency and locality by garbage collecting a smaller area more often, while exploiting typical lifetime characteristics to avoid undue overhead from long-lived objects.

1 Automatic Storage Reclamation

Garbage collection is the automatic reclamation of computer storage [Knu69, Coh81, App91]. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a "free" or "dispose" statement, garbage collected systems free the programmer from this burden. The garbage collector's function is to find data objects[1] that are no longer in use and make their space available for reuse by the the running program. An object is considered *garbage* (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a "dangling pointer" into a deallocated object.

This paper is intended to be an introductory survey of garbage collectors for uniprocessors, especially those developed in the last decade. For a more thorough treatment of older techniques, see [Knu69, Coh81].

1.1 Motivation

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies. A routine operating on a data structure should not have to know what other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be deallocated explicitly, some module must be responsible for knowing when *other* modules are not interested in a particular object.

[1] We use the term object loosely, to include any kind of structured data record, such as Pascal records or C structs, as well as full-fledged objects with encapsulation and inheritance, in the sense of object-oriented programming.

P. R. Wilson, "Uniprocessor garbage collection techniques", Proceedings of the International Workshop on Memory Management, St. Malo, France, September 1992. DOI: 10.1007/BFb0017182

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Basic Garbage Collection

- Mark-sweep
- Mark-compact
- Copying collectors

CC BY ND

Garbage Collection

- Avoid problems of reference counting and complexity of compile-time ownership tracking via **garbage collection**
  - Explicitly trace through allocated objects, recording which are in use; dispose of unused objects
  - Moves garbage collection to be a separate phase of the program's execution, rather than an integrated part of an objects lifecycle
  - Operation of the program (the **mutator**) and the garbage collector is interleaved
- Many tracing garbage collection algorithms exist:
  - Basic garbage collectors
    - Mark-sweep collectors
    - Mark-compact collectors
    - Copying collectors
  - Generational garbage collectors

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Mark-Sweep Collectors (1/4)

- Simplest automatic garbage collection scheme
- Two phase algorithm
  - Distinguish live objects from garbage (*mark*)
  - Reclaim the garbage (*sweep*)
  - Non-incremental: program is paused to perform collection when memory becomes tight

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Mark-Sweep Collectors (2/4)

- The **marking phase**: distinguishing live objects
  - Determine the **root set** of objects, comprising:
    - Any global variables
    - Any variable allocated on the stack, in any existing stack frame
  - Traverse the object graph staring at the root set to find other reachable objects
    - Starting from the root set, follow pointers to other objects
    - Follow every pointer in every object to systematically find all reachable objects
    - May proceed either breadth-first or depth-first
    - A cycle of objects that reference each other, but are not reachable from the root set, will not be marked
  - Mark reachable objects as alive
    - Set a bit in the object header, or in some separate table of live objects, to indicate that the object is reachable
    - Stop traversal at previously seen objects to avoid looping forever

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Mark-Sweep Collectors (3/4)

- The **sweep phase**: reclaiming objects that no longer live
  - Pass through entire heap once, examining each object for liveness
    - If marked as alive, keep the object
    - Otherwise, free the memory and reclaim the object's space

- When objects are reclaimed, their memory is marked as available
  - The system maintains a **free list** of blocks of unused memory
  - New objects are allocated in now unused memory if they fit; or in not-yet-used memory elsewhere on the heap
  - Fragmentation is a potential concern – but no worse than a language using **`malloc()`**/**`free()`**

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

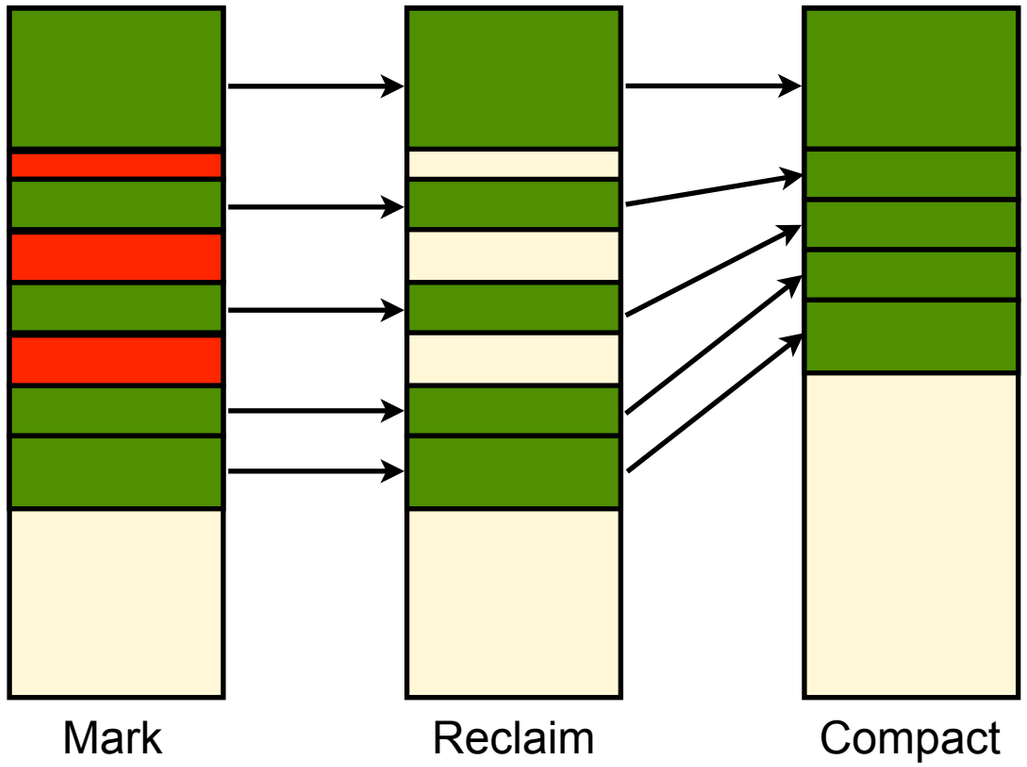
Mark-Sweep Collectors (4/4)

- Mark-sweep collectors are simple, but inefficient:
  - Garbage collection is slow and has unpredictable duration
    - Program is stopped while the collector runs
    - Time to collect garbage is unpredictable, and depends on the number of live objects (time for the marking phase) and size of the heap (time to sweep up unused objects)
    - Unlike reference counting, mark-sweep garbage collection is slower if the program has lots of memory allocated
  - Garbage collection has no locality of reference
    - Passing through the entire heap in unpredictable order disrupts operation of cache and virtual memory subsystem
    - Objects located where they fit, rather than where maintains locality of reference
  - Fragmentation of free space is a concern
    - Since objects are not moved, space may become fragmented, making it difficult to allocate large objects even though space available overall

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Mark-Compact Collectors (1/2)

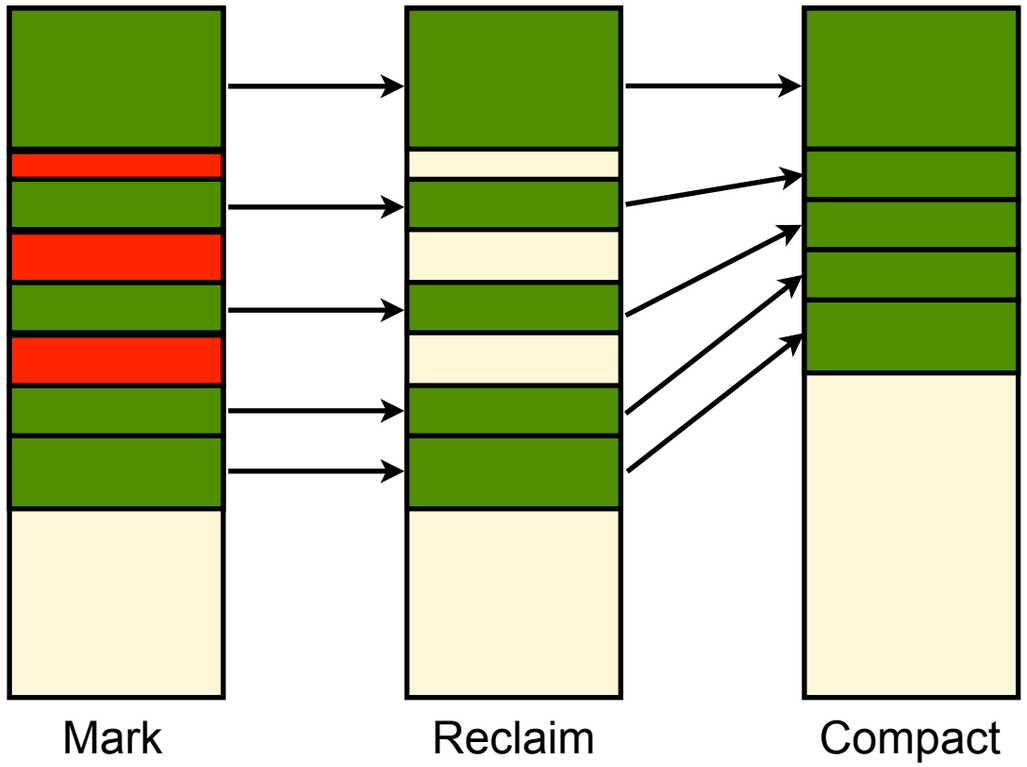
- Goal: solve fragmentation problems and speed-up memory allocation, compared to mark-sweep collectors
- Three logical phases:
  - Traverse object graph, **mark** live objects
  - **Reclaim** unreachable objects
  - **Compact** live objects, moving them to leave contiguous free space
- Reclaiming and compacting memory can be done in one pass, but still touches the entire address space



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Mark-Compact Collectors (2/2)

- Advantages:
  - Solves fragmentation problems – all free space is in one contiguous block
  - Allocation is very fast – always allocating from the start of the free block, so allocation is just incrementing pointer to start of free space and returning previous value
- Disadvantages:
  - Collection is slow, due to moving objects in memory, and time taken is unpredictable
  - Collection has poor locality of reference
  - Collection is complex – needs to update all pointers to moved objects



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

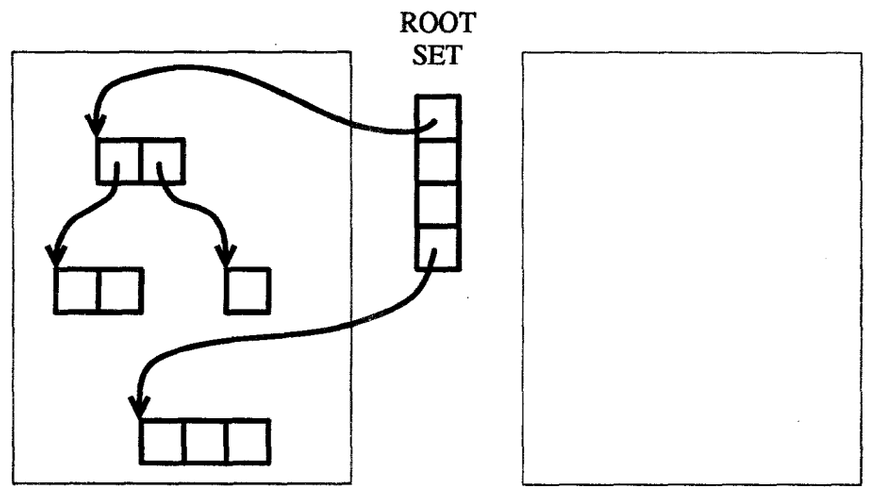
Copying Collectors (1/5)

- Copying collectors integrate the traversal (marking) and copying phases into one pass
  - All the live data is copied into one region of memory
  - All the remaining memory contains garbage, or has not yet been used
- Similar to mark-compact, but more efficient
- Time taken to collect is proportional to the number of live objects

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Copying Collectors (2/5)

- Stop-and-copy using semispaces:
  - Divide the heap into two halves, each one a contiguous block of memory
  - Allocations made linearly from one half of the heap only
    - Memory is allocated contiguously, so allocation is fast (as in the mark-compact collector)
    - No problems with fragmentation due to allocating data of different sizes
- When an allocation is requested that won't fit into the active half of the heap, a collection is triggered

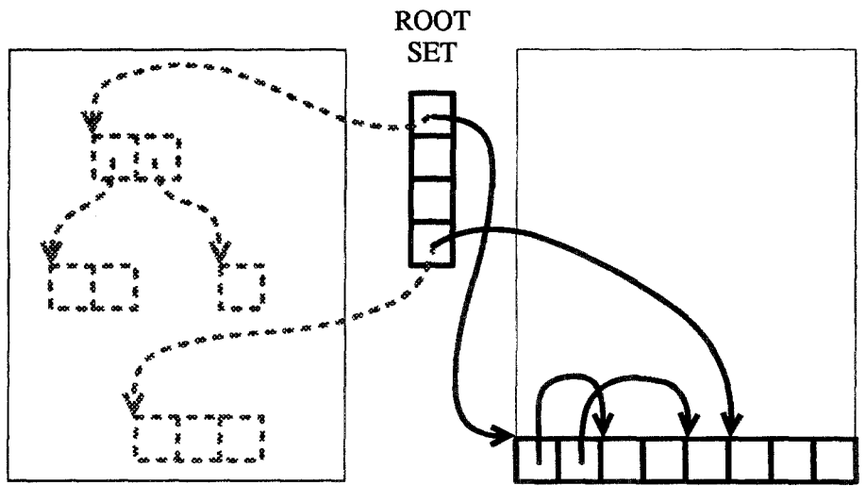


Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI 10.1007/BFb0017182

cc BY ND

Copying Collectors (3/5)

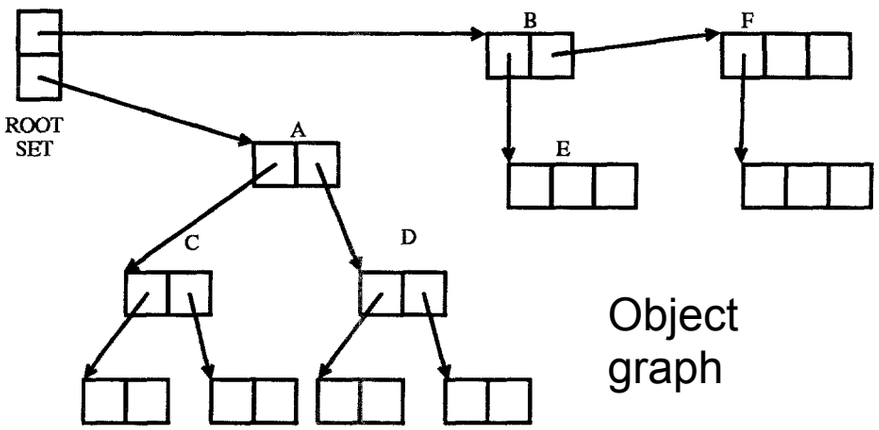
- Stop-and-copy using semispaces:
  - Collection stops execution of the program
  - A pass is made through the active space, and all live objects are copied to the other half of the heap
    - The Cheney algorithm is commonly used to make the copy in a single pass
    - Anything not copied is unreachable, and is simply ignored (and will eventually be overwritten by a later allocation phase)
  - The program is then restarted, using the other half of the heap as the active allocation region
- The role of the two parts of the heap (the two semispaces) reverses each time a collection is triggered



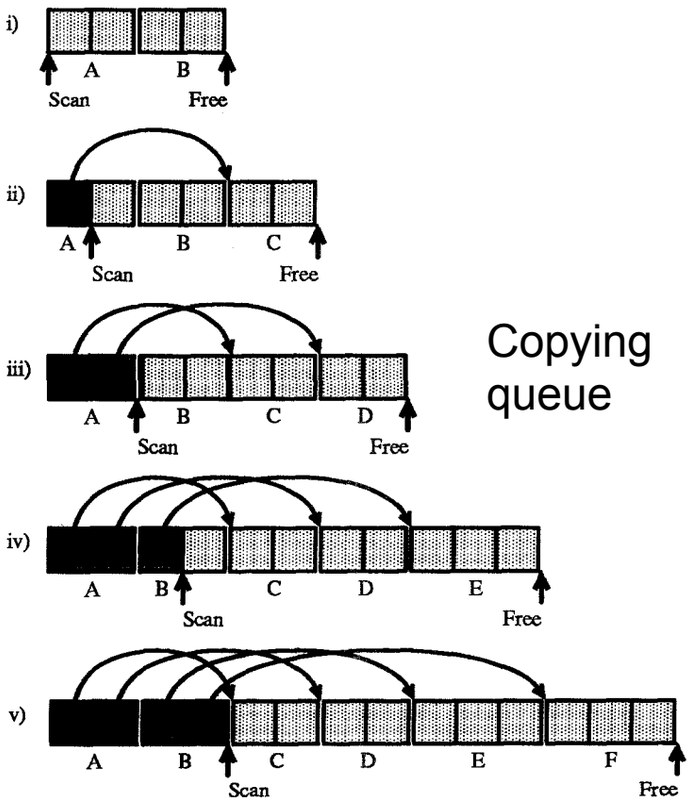
Source: P. Wilson, “Uniprocessor garbage collection techniques”, Proc IWMM’92, DOI 10.1007/BFb0017182

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Copying Collectors (4/5)



Object graph



Copying queue

Source: P. Wilson, “Uniprocessor garbage collection techniques”, Proc IWMM’92, DOI 10.1007/BFb0017182

- The Cheney Algorithm: breadth-first copying
  - A queue is created, to hold the set of live objects to be copied
  - The **root set** of objects, comprising global variables and all stack allocated variables, is found and added to the queue
  - Objects in the queue are examined in turn:
    - Any unprocessed objects they reference are added to the end of the queue
    - The object in the queue is then copied into the other semispace, and the original is marked as having been processed (pointers are updated as the copy is made)
- Once the end of the queue is reached, all live objects have been copied

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Copying Collectors (5/5)

- Efficiency of copying collectors:
  - Time taken for garbage collection depends on the amount of data copied, which depends on the number of live objects
  - Collection only happens when a semispace is full
- **If most objects die young**, can trade-off collection time *vs.* memory usage by increasing the size of the semispaces
  - A larger semispace takes longer to fill, so increases the how long objects need to live before they're copied
  - If most objects die young, most aren't copied
  - Uses more memory, but spends less time copying data

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Summary: Basic Garbage Collection

- Mark-sweep, mark-compact, and copying collectors have similar cost:
  - **Differ in where the cost is paid**: at time of allocation or time of collection; in memory usage or in processing time

- The mark-compact and copying algorithms move data, so cannot be used with languages that cannot unambiguously identify pointers
  - Can’t move an object, if you can’t be sure all pointers to it have been updated

cc BY ND

Generational Garbage Collection

CC BY ND

Object Lifetimes

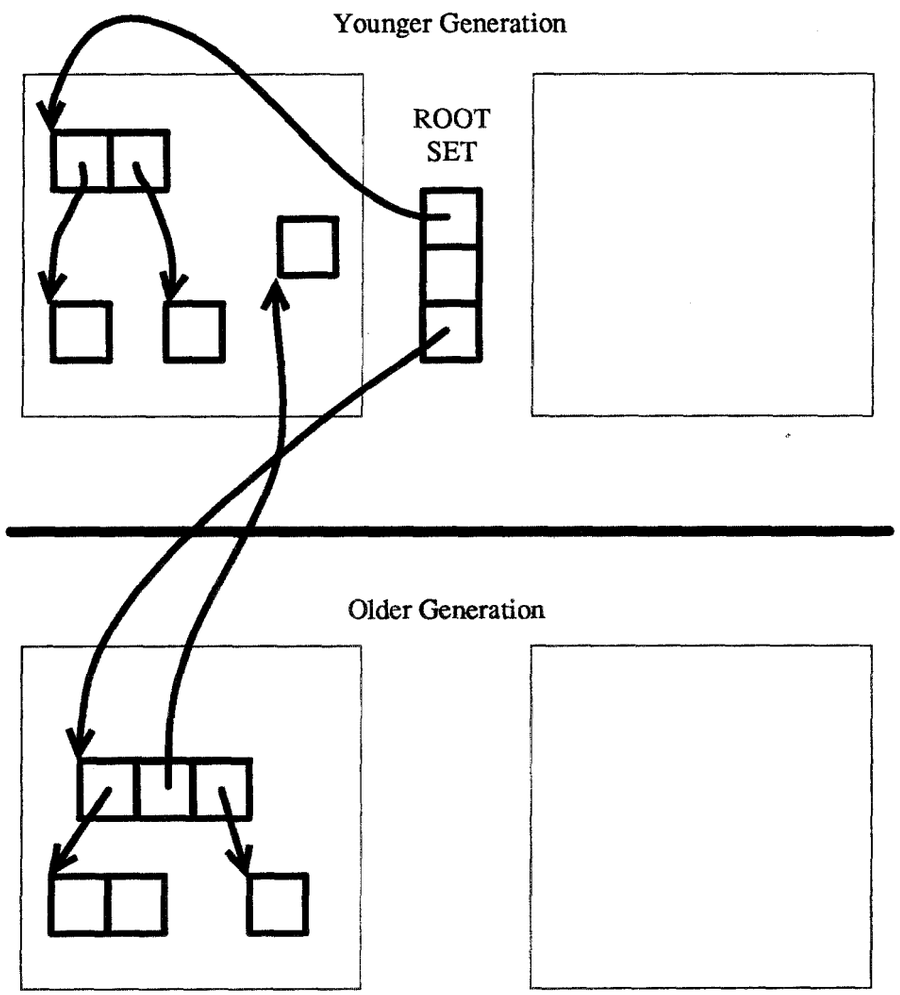
- Most objects have short time; a small percentage live much longer
  - This seems to be generally true, no matter what programming language is considered, across numerous studies
  - Although, obviously, different programs and different languages produce varying amount of garbage

- Implications:
  - When the garbage collector runs, live objects will be in a minority
  - Statistically, the longer an object has lived, the longer it is likely to live
  - Can we design a garbage collector to take advantage?

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Copying Generational Collectors (1/4)

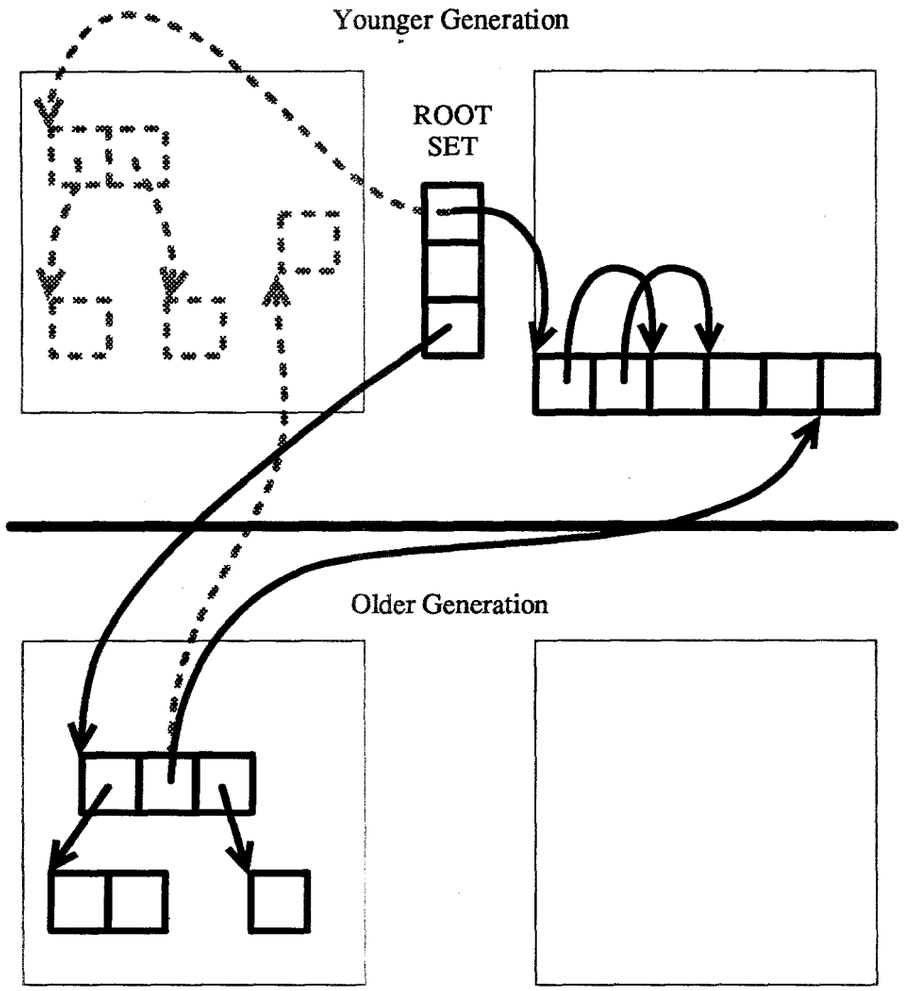
- In a generational garbage collector, the heap is split into regions for long-lived and young objects
  - Regions holding young objects are garbage collected more frequently
  - Objects are moved to the region for long-lived objects if they're still alive after several collections
  - More sophisticated approaches may have multiple generations, although the gains diminish rapidly with increasing numbers of generations
- Example: stop-and-copy using semispaces with two generations
  - All allocations occurs in the younger generation's region of the heap
  - When that region is full, collection occurs as normal
  - …



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Copying Generational Collectors (2/4)

- …
- Objects are tagged with the number of collections of the younger generation they have survived; if they're alive after some threshold, they're copied to the older generation's space during collection
- Eventually, the older generation's space is full, and is collected as normal



- Note: not to scale: older generations are generally much larger than the younger, as they're collected much less often

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Copying Generational Collectors (3/4)

- Young generation must collected independent of long-lived generation
- But – there may be references between generations
  - References from young objects to long-lived objects
    - Straight-forward – most young objects die before the long-lived objects are collected
    - Treat the younger generation objects as part of the root set for the long-lived generation, when collection of the long-lived generation is needed
  - References from long-lived objects to young objects:
    - Problematic, since requires scan of long-lived generation to detect
    - Maybe use indirection table (“pointers-to-pointers”) for references from long-lived generation to young generation
      - The indirection table forms part of the root set of the younger generation
      - Moving objects in younger generation requires updating indirection table, but not long-lived objects
      - Long-lived objects are collected infrequently, and may keep younger objects alive longer than expected

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Copying Generational Collectors (4/4)

- Variations on copying generational collectors are widely used
  - E.g., the HotSpot JVM uses a generational garbage collector

- Copying generational collectors are efficient:
  - Cost of collection is generally proportional to number of live objects
  - Most objects don’t live long enough to be collected; those that do are moved to a more rarely collected generation
  - Longer-lived generation must eventually be collected; this can be very slow

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Incremental Garbage Collection (1/5)

- Preceding discussion has assumed the collector “stops-the-world” when it runs
  - Problematic for interactive or real-time applications
- Desire a collector that can operate incrementally
  - Interleave small amounts of garbage collection with small runs of program execution
  - Implication: the garbage collector can’t scan the entire heap when it runs; must scan a fragment of the heap each time
  - Problem: the program (the “mutator”) can change the heap between runs of the garbage collector
  - Need to track changes made to the heap between garbage collector runs; be conservative and don’t collect objects that might be referenced – can always collect on the next complete scan

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

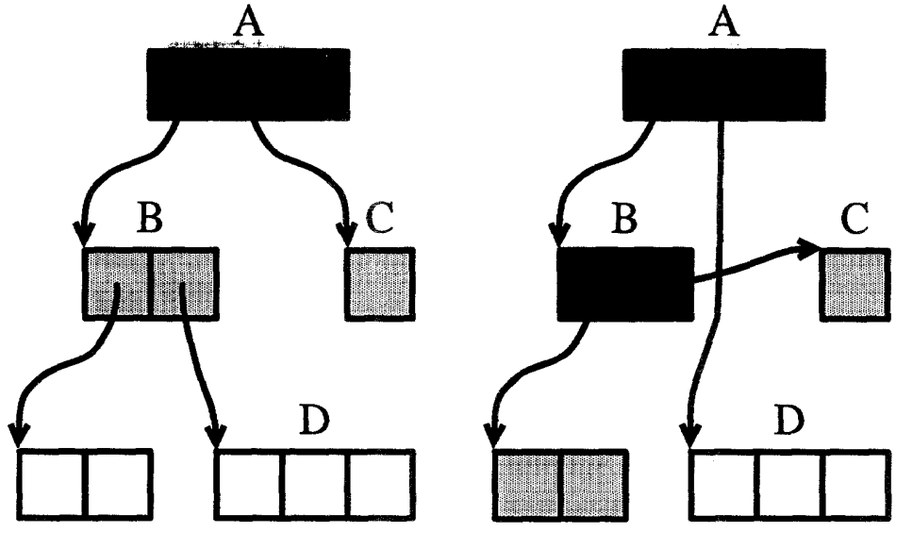
Incremental Garbage Collection (2/5)

- Tricolour marking: each object is labelled with a colour:
  - White – not yet checked
  - Grey – live, but some direct children not yet checked
  - Black – live
- Basic incremental collector operation:
  - Garbage collection proceeds with a wavefront of grey objects, where the collector is checking them, or objects they reference, for liveness
  - Black objects behind are behind the wavefront, and are known to be live
  - Objects ahead of the wavefront, not yet reached by the collection, are white; anything still white once all objects have been traced is garbage
  - No direct pointers from black objects to white – any program operation that will create such a pointer requires coordination with the collector

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Incremental Garbage Collection (3/5)

- Program and collector must coordinate
  - Garbage collector runs
    - Object A scanned, known to be live → black
    - Objects B and C are reachable via A, and are live, but some of their children have not been scanned → grey
    - Object D not checked → white
  - Program runs, and swaps the pointers from A→C and B→D such that A→D and B→C
- This creates a pointer from black to white
- Program must now coordinate with the collector, else collection will continue, marking object B black and its children grey, but D will not be reached since children of A have already been scanned

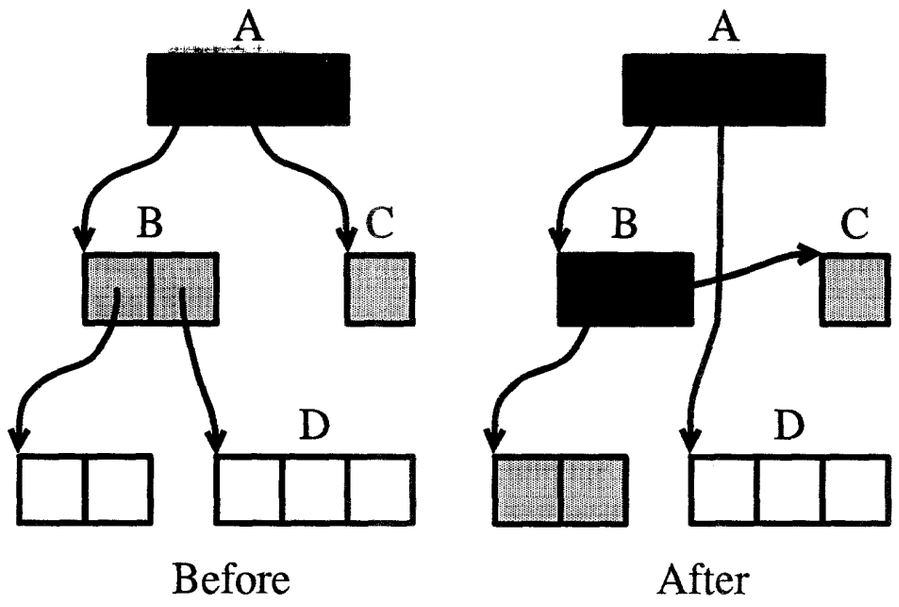


Before After

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Incremental Garbage Collection (4/5)

- Coordination strategies:
  - Read barrier: trap attempts by the program to read pointers to white objects, colour those objects grey, and let the program continue
    - Makes it impossible for the program to get a pointer to a white object, so it cannot make a black object point to a white
  - Write barrier: trap attempts to change pointers from black objects to point to white objects
    - Either then re-colour the black object as grey, or re-colour the white object being referenced as grey
    - The object coloured grey is moved onto the list of objects whose children must be checked



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Incremental Garbage Collection (5/5)

- Many variants on read- and write-barrier tricolour algorithms
  - Performance trade-off differs depending on hardware characteristics, and on the way pointers are represented
  - Write barrier generally cheaper to implement than read barrier, as writes are less common in most code
- There is a balance between collector operation and program operation
  - If the program tries to create too many new references from black to white objects, requiring coordination with the collector, the collection may never complete
  - Resolve by forcing a complete stop-the-world collection if free memory is exhausted, or after a certain amount of time

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Real-time Garbage Collection

- Real-time collectors build incremental collectors
  - Two basic approaches:
    - Work based: every request to allocate an object or assign an object reference does some garbage collection; amortise collection cost with allocation cost
    - Time based: schedule an incremental collector as a periodic task
  - Obtain timing guarantees by limiting amount of garbage that can be created in a given interval to less than that which can be collected
- The amount of garbage that can be collected can be measured: how fast can the collector scan memory (and copy objects, if a copying collector)
  - Cannot collect garbage faster than the collector can scan memory to determine if objects are free to be collected
  - This must be a worse-case collection rate, if the collector has varying runtime
- The programmer must bound the amount of garbage generated to within the capacity of the collector



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-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.
Copyright © 2003 ACM 1-58113-628-5/03/0001 $5.00.

Bacon *et al*., "A real-time garbage collector with low overhead and consistent utilization". ACM Symposium on Principles of Programming Languages, New Orleans, LA, USA, January 2003. DOI: 10.1145/604131.604155

CC BY ND

Practical Factors

- Memory Overheads
- Interaction with Virtual Memory
- Garbage Collection for Weakly-Typed Languages

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Memory Overheads

- Garbage collection trades ease-of-use for predictability and overhead
- Garbage collected programs will use significantly more memory than **correctly written** programs with manual memory management
  - Many copying collectors maintain two semispaces, so double memory usage
  - But – many programs with manual memory management are **not correct**

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Interaction with Virtual Memory

- Virtual memory subsystems page out unused data in an LRU manner
- Garbage collector scans objects, paging data back into memory
- Leads to thrashing if the working set of the garbage collector larger than memory
  - Open research issue: combining virtual memory with garbage collector

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Garbage Collection for Weakly-typed Languages

- Collectors rely on being able to identify and follow pointers, to determine what is a live object
- Weakly typed, such as C, can cast any integer to a pointer, and perform pointer arithmetic
  - Implementation-defined behaviour, since pointers and integers are not guaranteed to be the same size
  - Greatly complicates garbage collection:
    - Need to be conservative: any memory that might be a pointer must be treated as one
    - Boehm-Demers-Weiser collector can be used for C and C++ (http://www.hboehm.info/gc/) – works for strictly conforming ANSI C code, but beware that much code is not conforming

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Memory Management Trade-offs



- Rust pushes memory management complexity onto the programmer
  - Predictable run-time performance, low run-time overheads
  - Uniform resource management framework, including memory
  - Limits the programs that may be expressed – matches common patterns in good C code
- Garbage collection imposes run-time costs and complexity, but simpler for the programmer

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Summary

- Garbage collection
  - Mark-sweep
  - Mark-compact
  - Copying collectors
  - Generational algorithms
  - Incremental algorithms
- Real-time garbage collection
- Practical factors

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow