We make the figures from *The Garbage Collection Handbook: The Art of Automatic Memory Management*, Richard Jones, Antony Hosking, Eliot Moss (Chapman and Hall, Second edition, 2023) available for fair use by educators and students. We ask that the following credit be given:

Chapter 1

Introduction

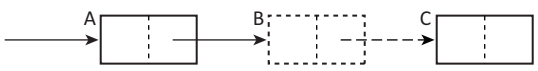


Figure 1.1: Premature deletion of an object may lead to errors. Here B has been freed. The live object A now contains a dangling pointer. The space occupied by C has leaked: C is not reachable but it cannot be freed.

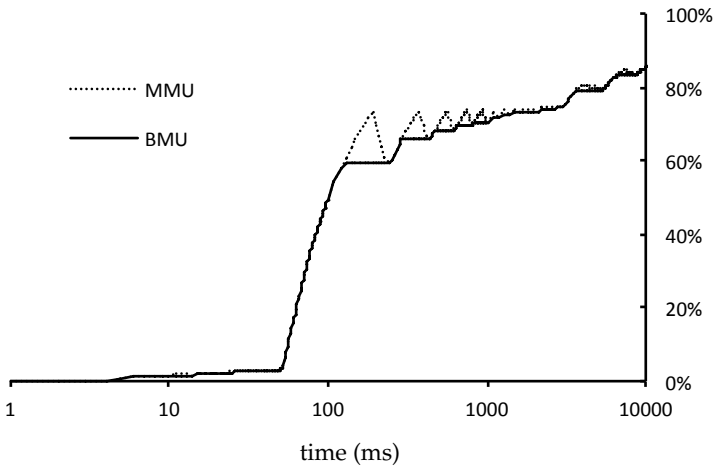


Figure 1.2: Minimum mutator utilisation and bounded mutator utilisation curves display concisely the (minimum) fraction of time spent in the mutator, for any given time window. MMU is the *minimum* mutator utilisation (y) in any time window (x) whereas BMU is the minimum mutator utilisation in that time 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.

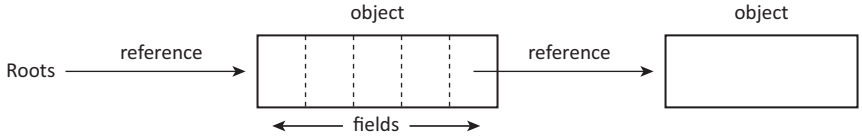


Figure 1.3: Roots, heap cells and references. Objects, denoted by rectangles, may be divided into a number of fields, delineated by dashed lines. References are shown as solid arrows.

Chapter 2

Mark-sweep garbage collection

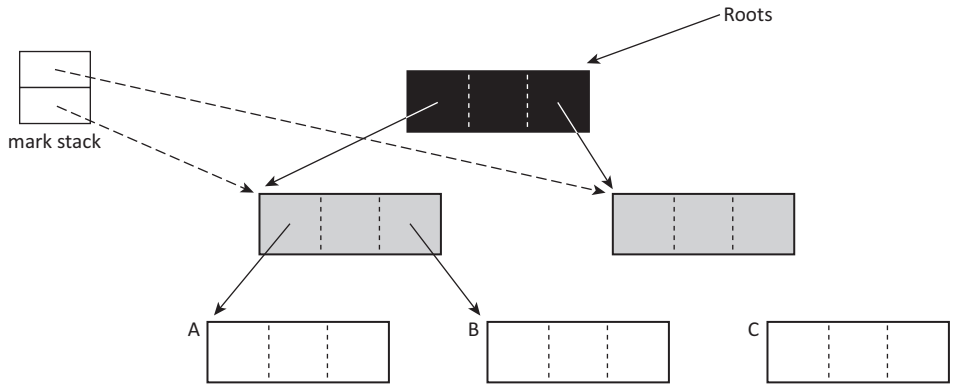


Figure 2.1: Marking with the tricolour abstraction. Black objects and their children have been processed by the collector. The collector knows of grey objects but has not finished processing them. White objects have not yet been visited by the collector (and some will never be).

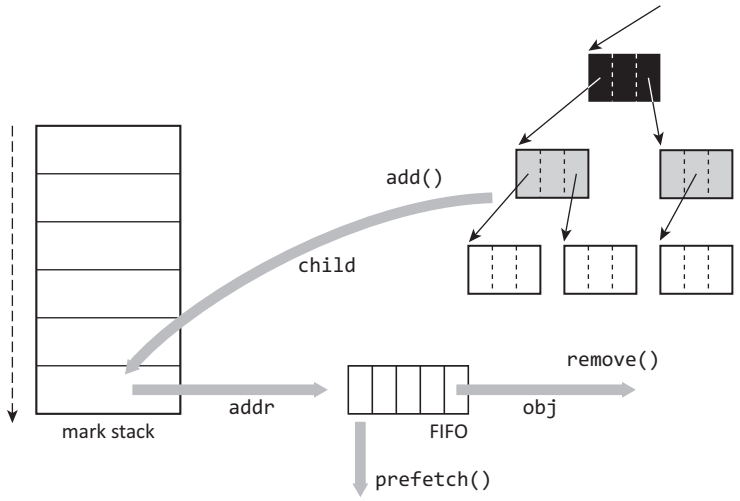


Figure 2.2: Marking with a FIFO prefetch buffer. As usual, references are added to the work list by being pushed onto the mark stack. However, to remove an item from the work list, the oldest item is removed from the FIFO buffer and the entry at the top of the stack is inserted into it. The object to which this entry refers is prefetched so that it should be in the cache by the time this entry leaves the buffer.

Chapter 3

Mark-compact garbage collection

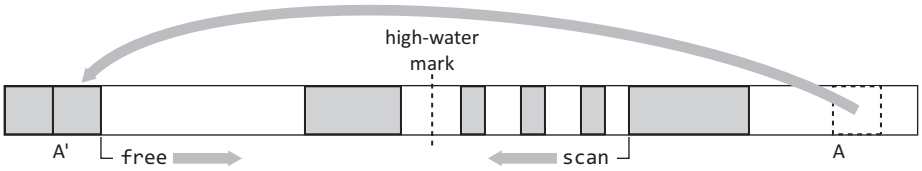
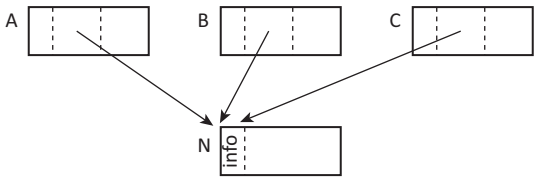
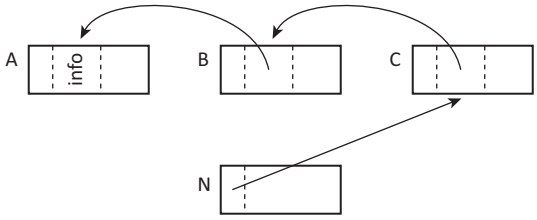


Figure 3.1: Edwards's Two-Finger algorithm. Live objects at the top of the heap are moved into free gaps at the bottom of the heap. Here, the object at A has been moved to A′. The algorithm terminates when the `free` and `scan` pointers meet.



(a) Before threading: three objects refer to N



(b) After threading: all pointers to N have been 'threaded' so that the objects that previously referred to N can now be found from N. The value previously stored in the header word of N, which is now used to store the threading pointer, has been (temporarily) moved to the first field (in A) that referred to N.

Figure 3.2: Threading pointers

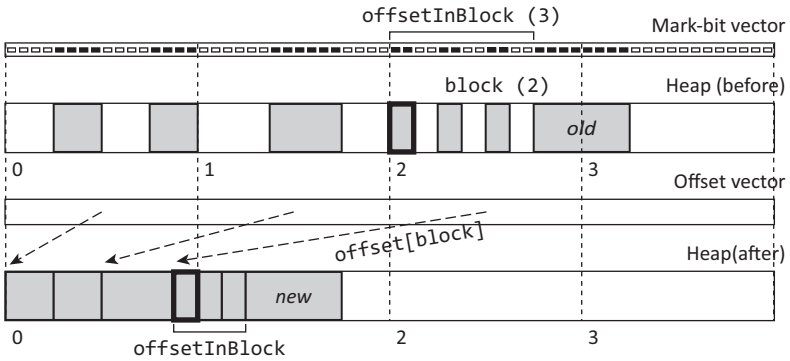
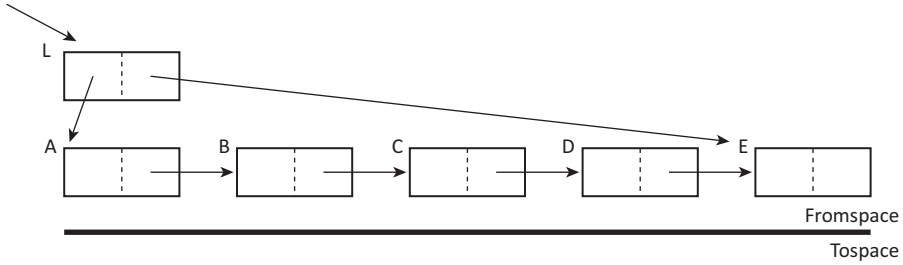


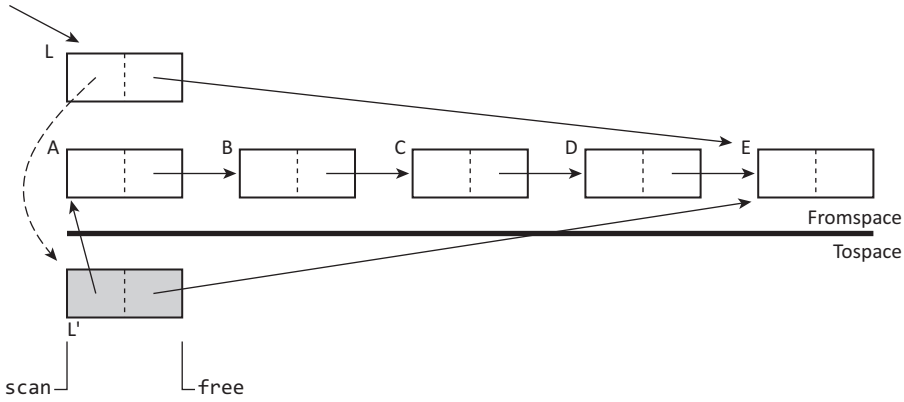
Figure 3.3: The heap (before and after compaction) and metadata used by Compressor [Kermany and Petrank, 2006]. Bits in the mark bit vector indicate the start and end of each live object. Words in the offset vector hold the address to which the first live object that starts in their corresponding block will be moved. Forwarding addresses are not stored but are calculated when needed from the offset and mark bit vectors.

Chapter 4

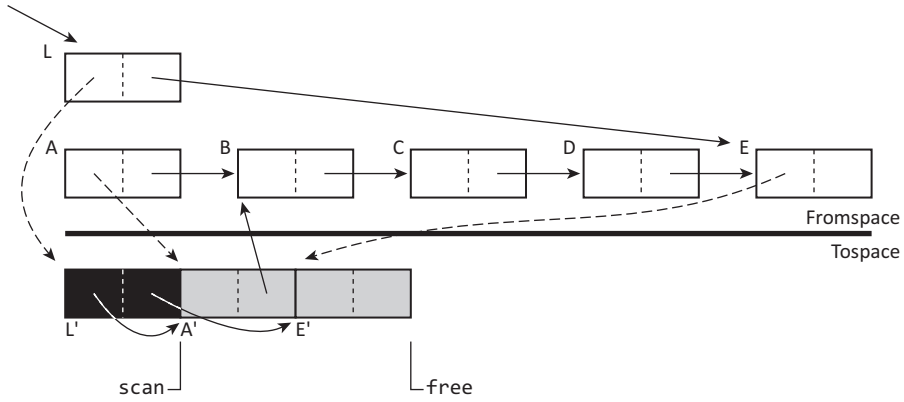
Copying garbage collection



(a) Fromspace after flip but before tracing

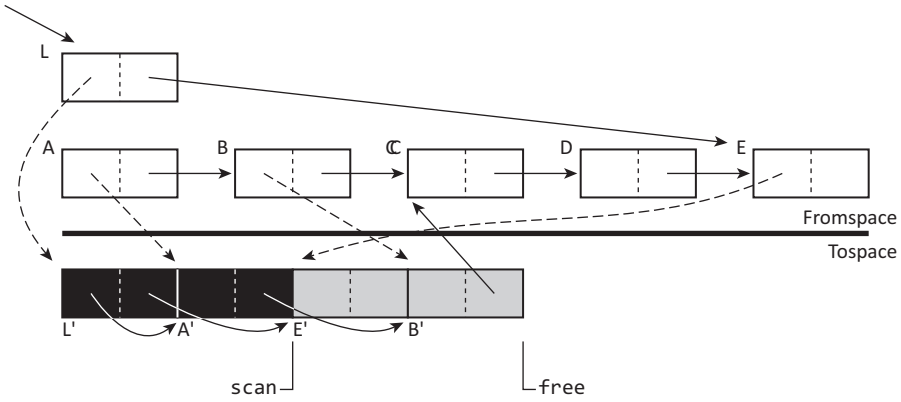


(b) Copy the root, L

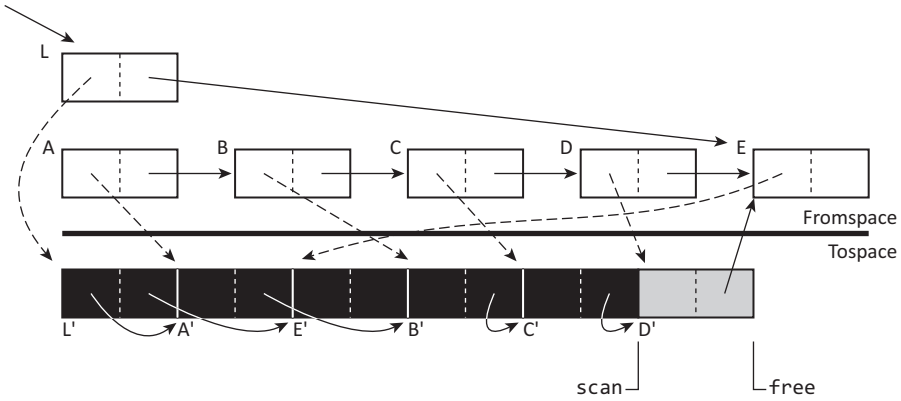


(c) Scan L's replica

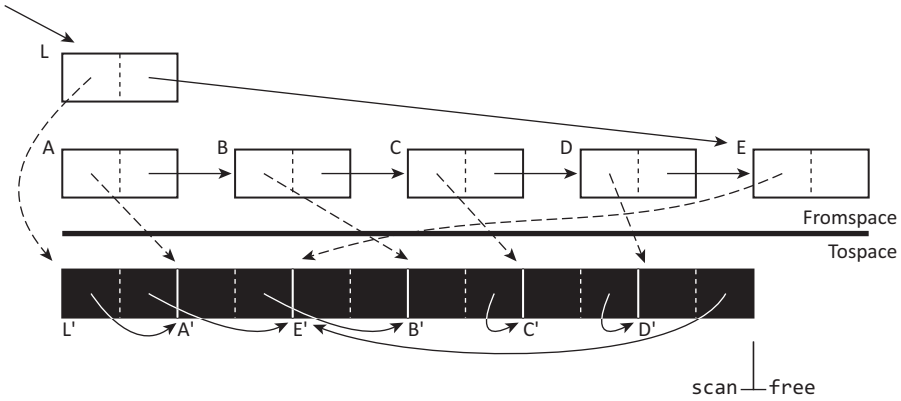
Figure 4.1: Cheney copying garbage collection: an example



(d) Scan A's replica, and so on...

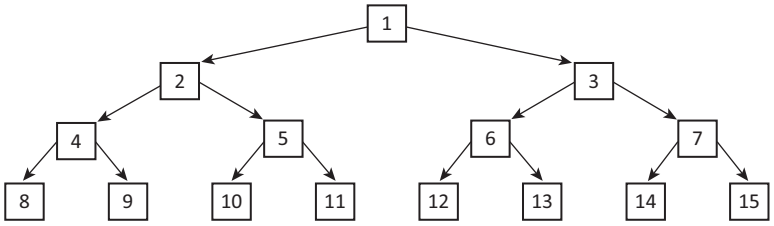


(e) Scan C's replica.



(f) Scan D's replica. `scan=free` so collection is complete.

Figure 4.1 (continued): Cheney copying garbage collection: an example



(a) The tree to copy

Depth-first	1	2	4	8	9	5	10	11	3	6	12	13	7	14	15
Breadth-first	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hierarchical decomposition	1	2	3	4	8	9	5	10	11	6	12	13	7	14	15
Online object reordering	***1***	***2***	***3***	***7***	***5***	***11***	10	4	15	14	6	***13***	9	8	12

(b) Placement of objects in the heap after copying

Figure 4.2: Copying a tree with different traversal orders. Each row shows how a traversal order lays out objects in tospace, assuming that three objects can be placed on a page (indicated by the thick borders). For *online object reordering*, prime numbered (bold italic) fields are considered to be hot.

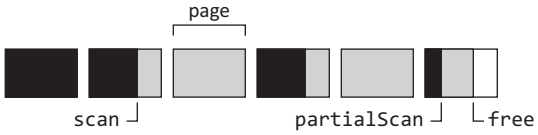


Figure 4.3: Moon's approximately depth-first copying. Each block represents a page. As usual, scanned fields are black, and copied but not yet scanned ones are grey. Free space is shown in white.

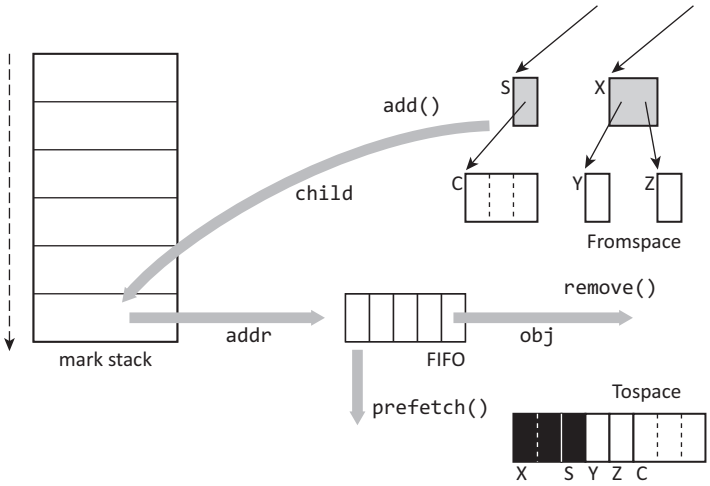


Figure 4.4: A FIFO prefetch buffer (discussed in Chapter 2) does not improve locality with copying as distant cousins (C, Y, Z), rather than parents and children, tend to be placed together.

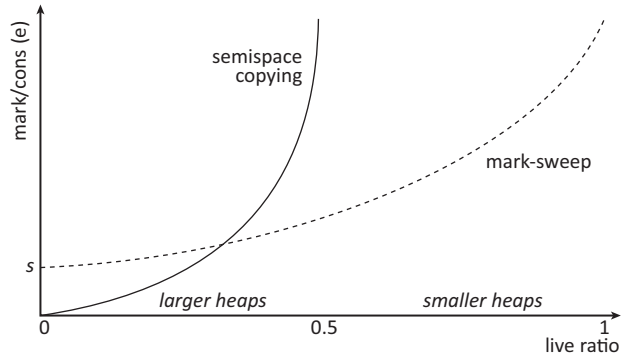


Figure 4.5: Mark/cons ratios for mark-sweep and copying collection (lower is better)

Chapter 5

Reference counting

Figure 5.1: Deferred reference counting schematic, showing whether reference counting operations on pointer loads or stores should be deferred or be performed eagerly. The arrows indicate the source and target of pointers loaded or stored.

Copyrighted figure withheld

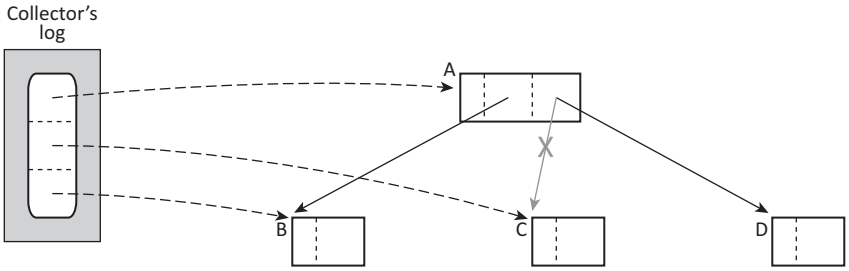
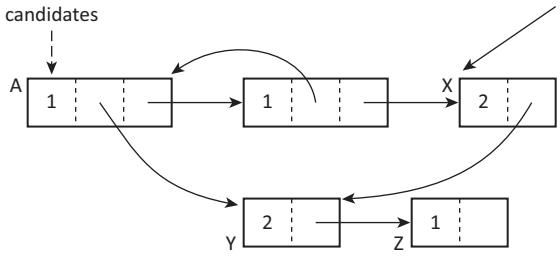
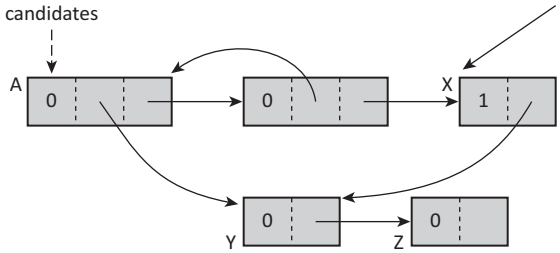


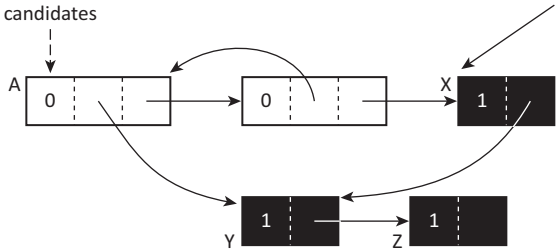
Figure 5.2: Coalesced reference counting: if A was modified in the previous epoch, for example by overwriting the reference to C with a reference to D, A's reference fields will have been copied to the log. The old referent C can be found in the collector's log and the most recent new referent D can be found directly from A.



(a) Before `markGrey`.



(b) After `markGrey`, all objects reachable from a candidate object have been marked grey and the effect of references internal to this grey subgraph have been removed. Note that X, which is still reachable, has a non-zero reference count.



(c) After `scan`, all reachable objects are black and their reference counts have been corrected to reflect live references.

Figure 5.3: Cyclic reference counting. The first field of each object is its reference count.

Figure 5.4: The synchronous Recycler state transition diagram, showing mutator and collector operations and the colours of objects

Chapter 6

Comparing garbage collectors

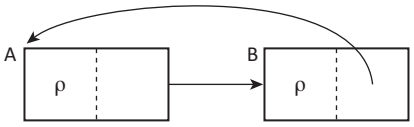
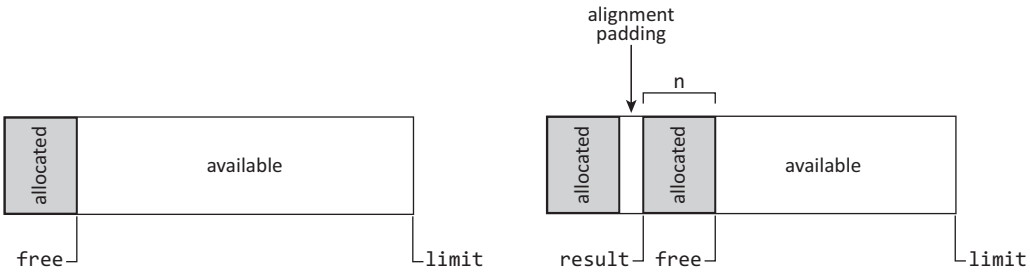


Figure 6.1: A simple cycle

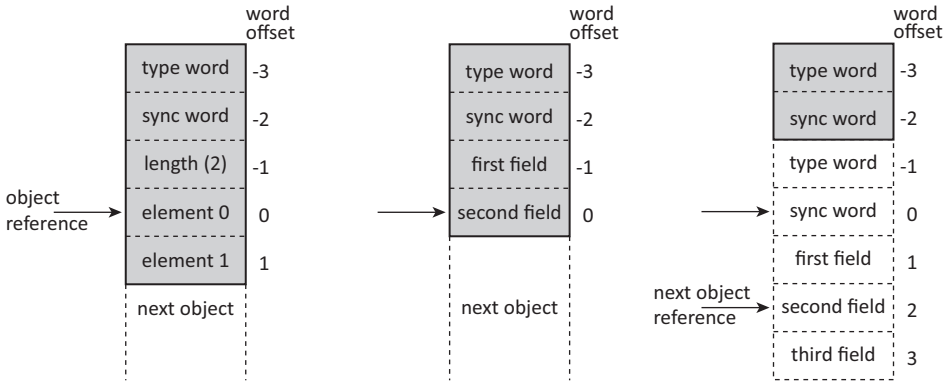


(a) Before (b) After

Figure 7.1: Sequential allocation: a call to `sequentialAllocate(n)` which advances the `free` pointer by the size of the allocation request, `n`, plus any padding necessary for proper alignment.

Chapter 7

Allocation



(a) Array (b) Scalar (non-Array) (c) No-Fields

Figure 7.2: A Java object header design for heap parsability. Grey indicates the words forming the referent object. Neighbouring objects are shown with dashed lines.

Chapter 8

Partitioning the heap

Chapter 9

Generational garbage collection

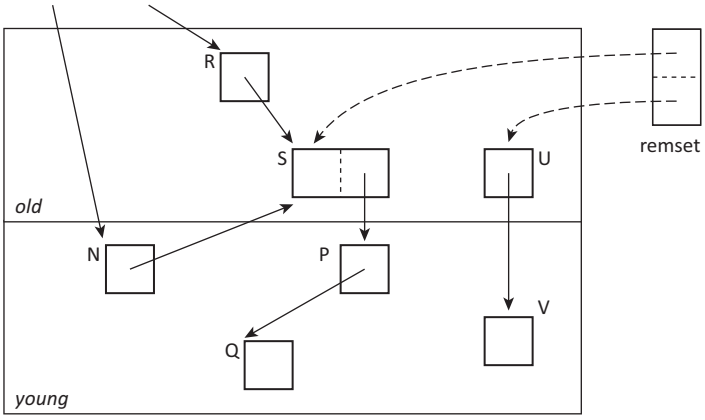
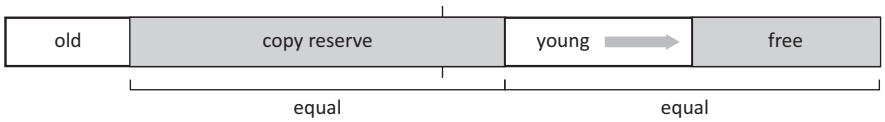


Figure 9.1: Inter-generational pointers. If live objects in the young generation are to be preserved without tracing the whole heap, a mechanism and a data structure are needed to remember objects S and U in the old generation that hold references to objects in the young generation.

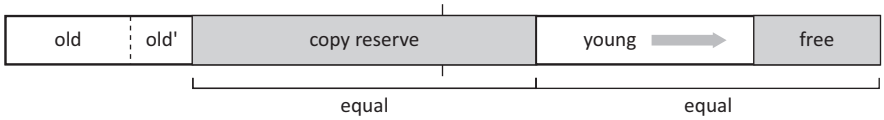
Figure 9.2: Survival rates with a copy count of 1 or 2. The curves show the fraction of objects that will survive a future collection if they were born at time x. Curve (b) shows the proportion that will survive one collection and curve (c) the proportion that will survive two. The coloured areas show the proportions of objects that will not be copied or will be promoted (copied) under different copy count regimes.

Figure 9.3: Shaw's bucket brigade system. Objects are copied within the young generation from a creation space to an aging semispace. By placing the aging semispace adjacent to the old generation at even numbered collections, objects can be promoted to the old generation simply by moving the boundary between generations.

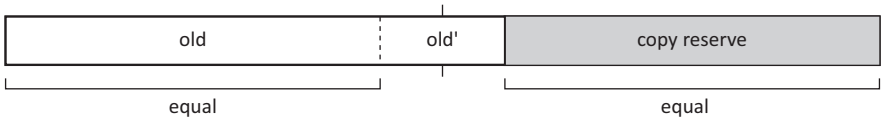
Figure 9.4: High water marks. Objects are copied from a fixed creation space to an aging semispace within a younger generation and then promoted to an older generation. Although all survivors in an aging semispace are promoted, by adjusting a 'high water mark', we can choose to copy or promote an object in the creation space simply through an address comparison.



(a) Before a minor collection, the copy reserve must be at least as large as the young generation.

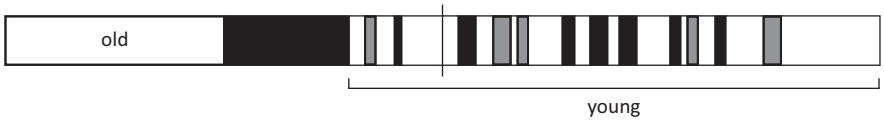


(b) At a minor collection, survivors are copied into the copy reserve, extending the old generation. The copy reserve and young generation are reduced but still of equal size.

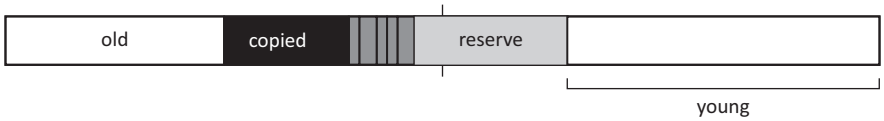


(c) After a minor collection and before a major collection. Only objects in the oldest region, old, will be evacuated into the copy reserve. After the evacuation, all live old objects can be moved to the beginning of the heap.

Figure 9.5: Appel's simple generational collector. Grey areas are empty.



(a) Objects copied or marked



(b) Marked objects compacted

Figure 9.6: Switching between copying and marking the young generation. (a) The copy reserve is full. Black objects from the young generation have been copied into the old generation. Grey objects have been marked but not copied. All other new objects are dead. (b) The compaction pass has slid the grey objects into the old generation.

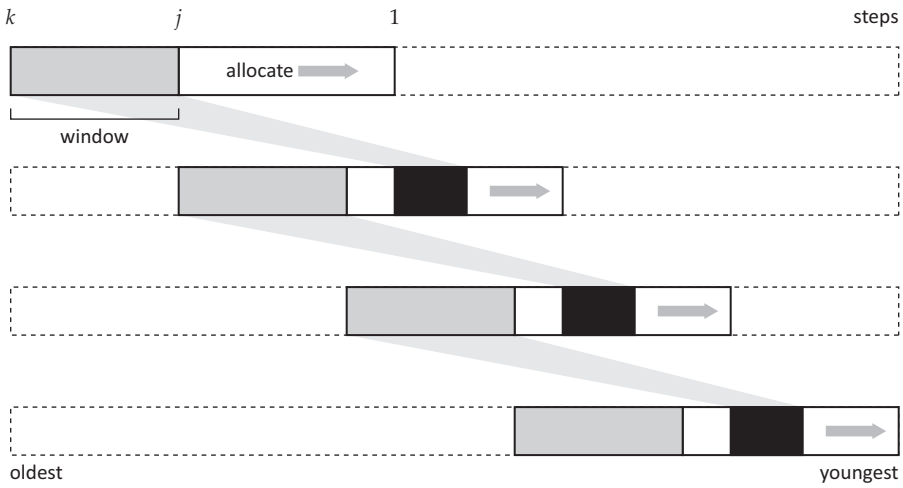


Figure 9.7: Renewal-Older-First garbage collection. At each collection, the objects least recently collected are scavenged and survivors are placed after the youngest objects.

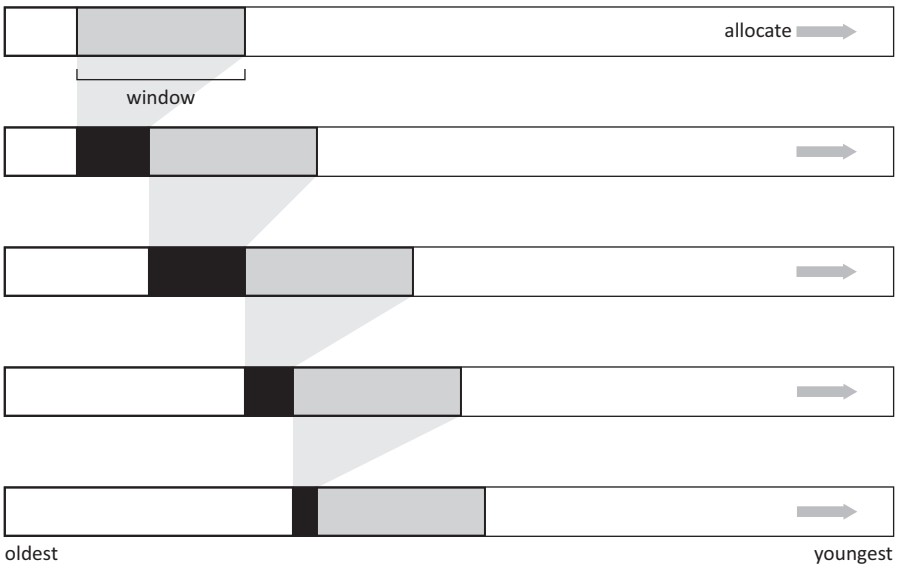


Figure 9.8: Deferred-Older-First garbage collection. A middle-aged window of the heap is selected for collection. Survivors are placed after the survivors of the previous collection. The goal is that the collector will discover a sweet spot, where the survival rate is very low and the window advances very slowly.

Figure 9.9: Beltway can be configured as any copying collector. Each figure shows the increment used for allocation, the increment to be collected and the increment to which survivors will be copied for each configuration.

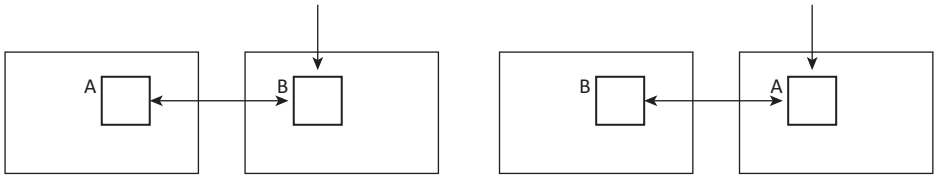
Chapter 10

Other partitioned schemes

Figure 10.1: The Treadmill collector: objects are held on a double-linked list. Each of the four segments holds objects of a different colour, so that the colour of an object can be changed by 'unsnapping' it from one segment and 'snapping' it into another. The pointers controlling the Treadmill are the same as for other incremental copying collectors [Baker, 1978]: scanning is complete when `scan` meets `T`, and memory is exhausted when `free` meets `B`.

Figure 10.2: The Train copying collector

Copyrighted figure withheld



(a) Before collecting the first car (b) Before collecting the next car

Figure 10.3: A 'futile' collection. After a collection which moves A to a fresh car, the external reference is updated to refer to A rather than B. This presents the same situation to the collector as before, so no progress can be made.

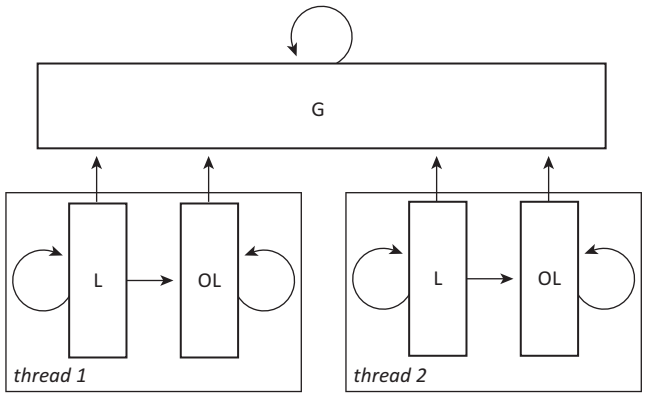


Figure 10.4: Thread-local heaplet organisation, indicating permitted pointer directions between purely local (L), optimistically-local (OL) and shared heaplets (G) [Jones and King, 2005]

Figure 10.5: A continuum of tracing collectors. Spoonhower *et al.* contrast an evacuation threshold — sufficient live data to make a block a candidate for evacuation — with an *allocation threshold* — the fraction of a block's free space reused for allocation.

Copyrighted figure withheld

Figure 10.6: Incremental incrementally compacting garbage collection. One space (fromspace) is chosen for evacuation to an empty space (tospace), shown as grey; the other spaces are collected in place. By advancing the two spaces, the whole heap is eventually collected.

Copyrighted figure withheld

Figure 10.7: Allocation in immix, showing *blocks* of *lines*. Immix uses bump pointer allocation within a partially empty block of small objects, advancing `lineCursor` to `lineLimit`, before moving onto the next group of unmarked lines. It acquires wholly empty blocks in which to bump-allocate medium-sized objects. Immix marks both objects and lines. Because a small object may span two lines (but no more), immix treats the line after any sequence of (explicitly) marked lines as implicitly marked: the allocator will not use it.

Figure 10.8: Mark-Copy divides the space to be collected into blocks. After the mark phase has constructed a remembered set of objects containing pointers that span blocks, the blocks are evacuated and unmapped, one at a time.

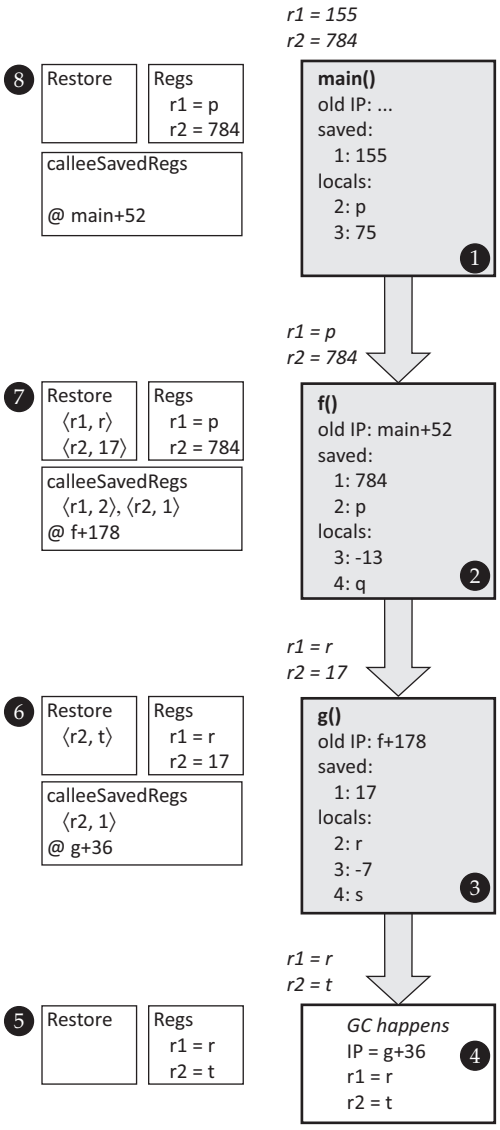
Figure 10.9: Ulterior reference counting schematic: the heap is divided into a space that is managed by reference counting and one that is not. The schematic shows whether reference counting operations on pointer loads or stores should be performed eagerly, deferred or ignored.

Chapter 11

Run-time interface

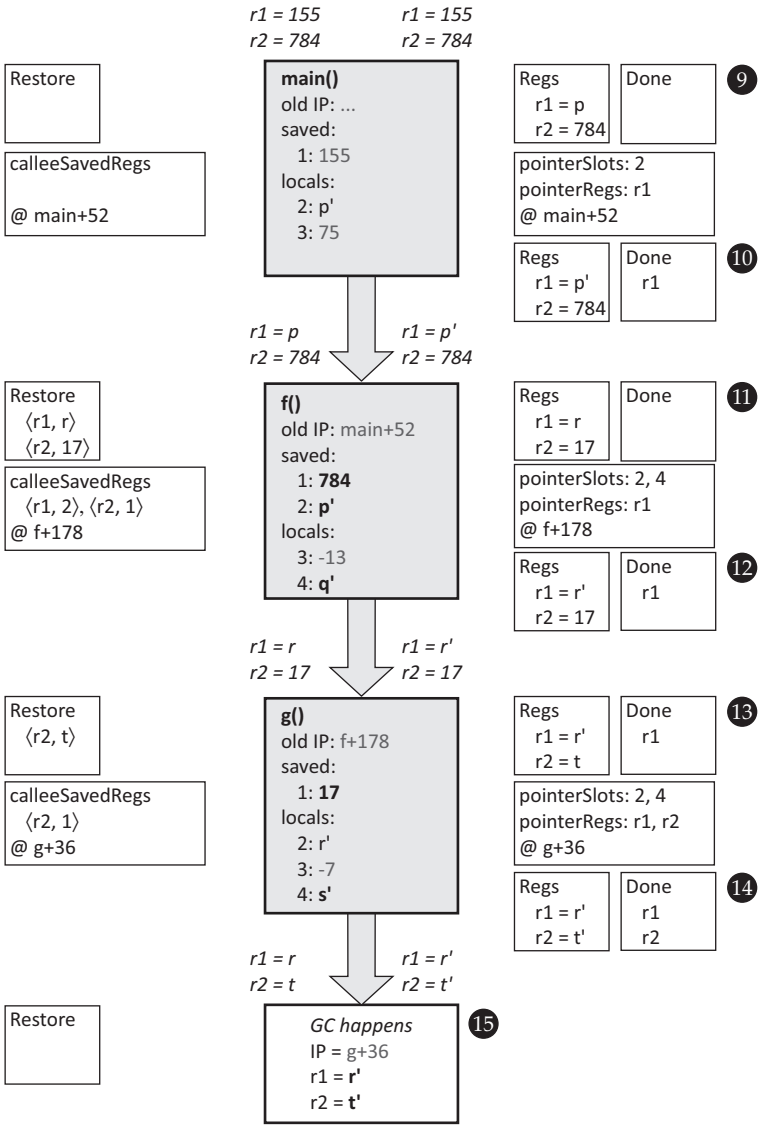
Figure 11.1: Conservative pointer finding. The two-level search tree, block header and map of allocated blocks in the Boehm-Demers-Weiser conservative collector.

Copyrighted figure withheld



(a) Stack scanning: walking from the top

Figure 11.2: Stack scanning



(b) Stack scanning: walking back to the top

Figure 11.2 (continued): Stack scanning

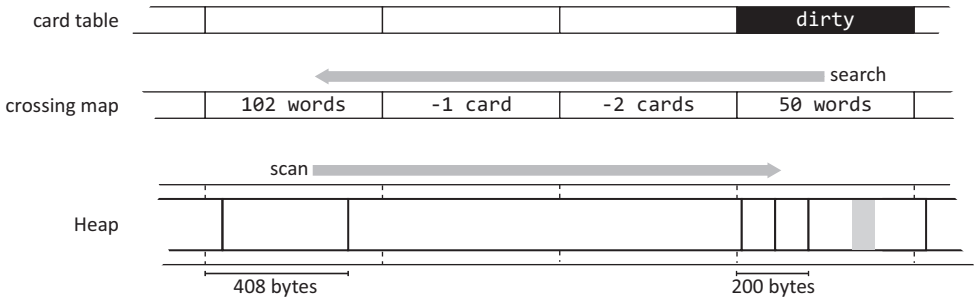


Figure 11.3: Crossing map with slot remembering card table. One card has been dirtied (shown in black). The updated field is shown in grey. The crossing map shows offsets (in words) to the last object in a card.

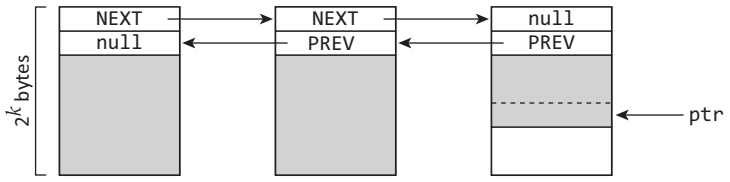


Figure 11.4: A stack implemented as a chunked list. Shaded slots contain data. Each chunk is aligned on a 2^k byte boundary.

Chapter 12

Language-specific concerns

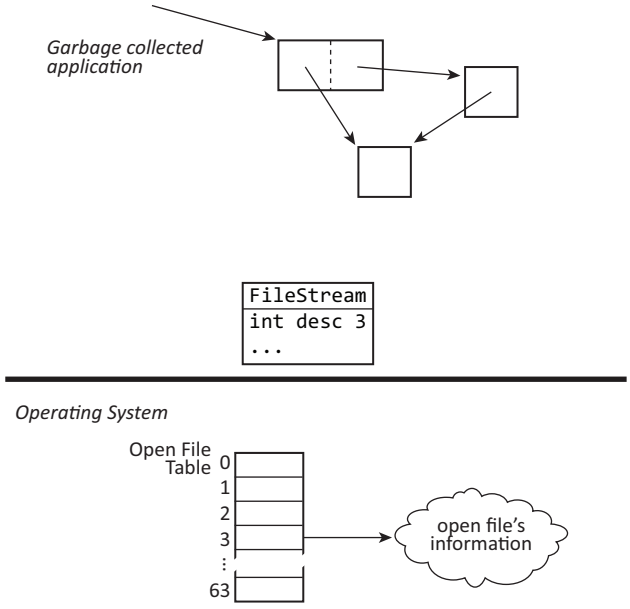


Figure 12.1: Failure to release a resource: a `FileStream` object has become unreachable, but its file descriptor has not been closed.

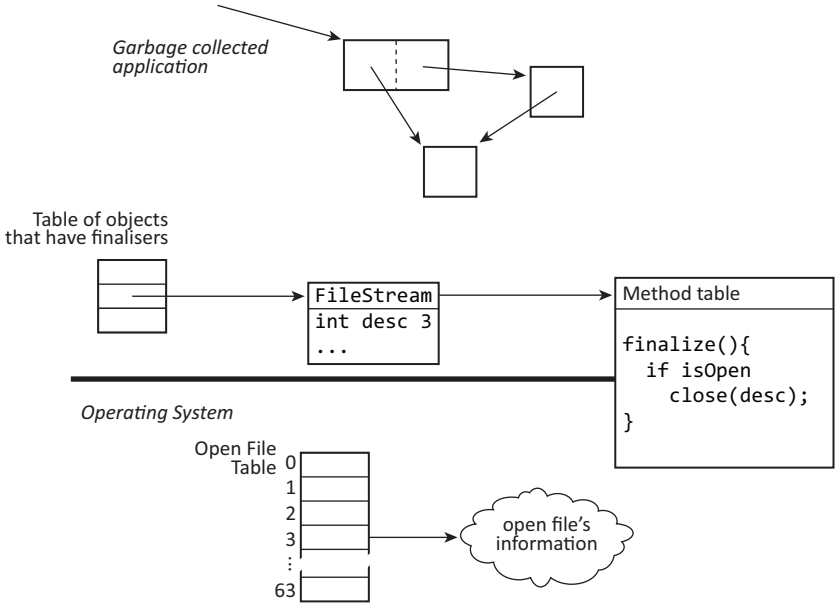


Figure 12.2: Using a finaliser to release a resource: here, an unreachable `FileStream` object has a finaliser to close the descriptor.

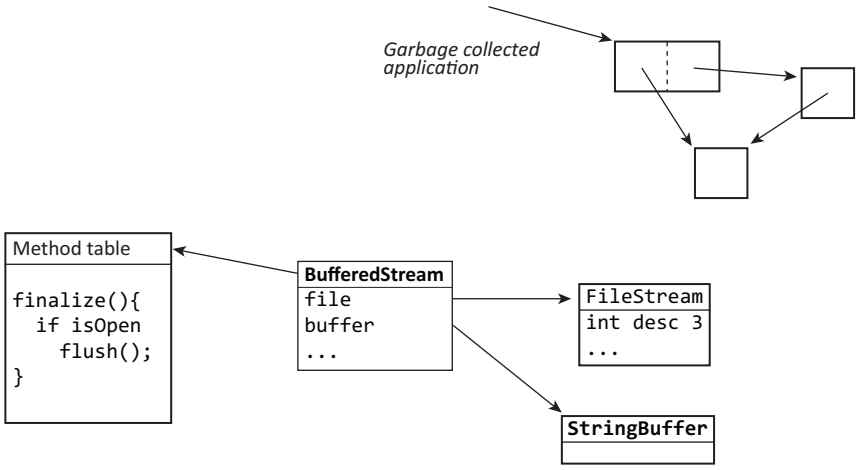
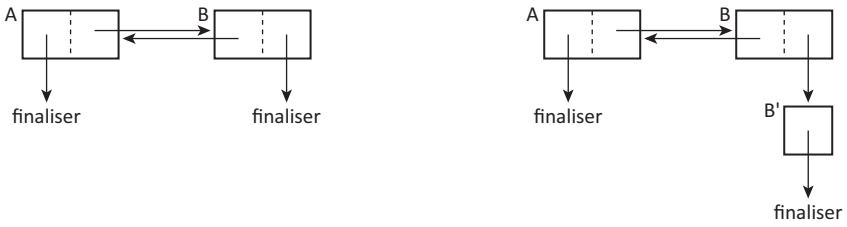


Figure 12.3: Object finalisation order. Unreachable `BufferedStream` and `FileStream` objects, which must be finalised in that order.



(a) Original order (b) Refactored to handle cycles

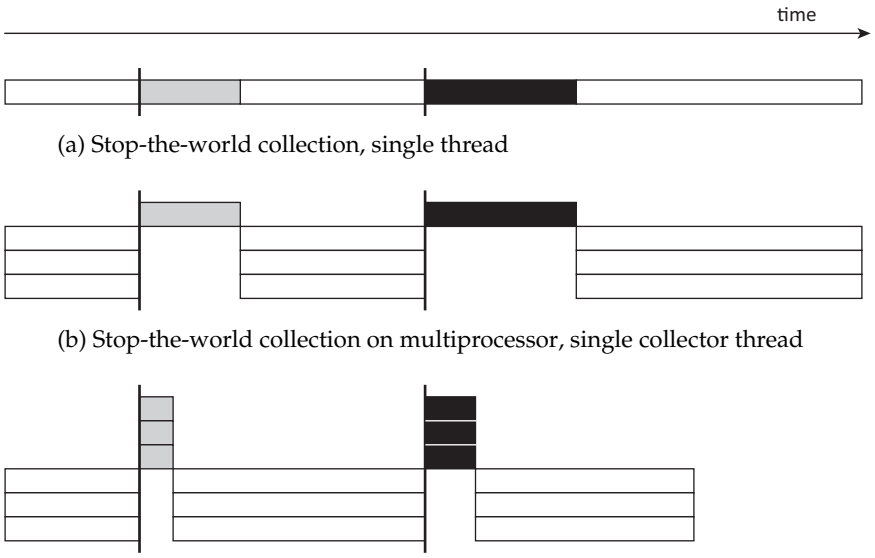
Figure 12.4: Restructuring to force finalisation order in cyclic object graphs

Chapter 13

Concurrency preliminaries

Chapter 14

Parallel garbage collection



(a) Stop-the-world collection, single thread

(b) Stop-the-world collection on multiprocessor, single collector thread

(c) Stop-the-world parallel collection

Figure 14.1: Stop-the-world collection: each bar represents an execution on a single processor. The coloured regions represent different collection cycles.

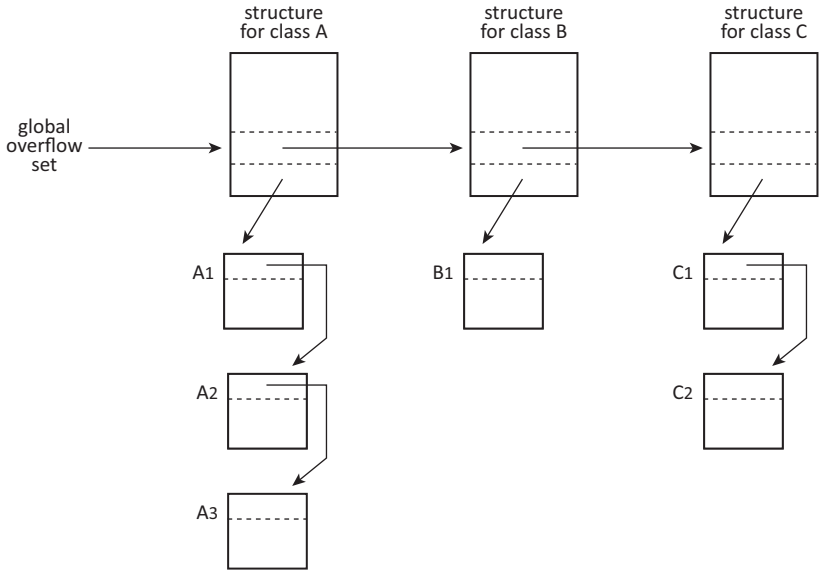


Figure 14.2: Global overflow set implemented as a list of lists [Flood *et al.*, 2001]. The class structure for each Java class holds the head of a list of overflow objects of that type, linked through the class pointer field in their header.

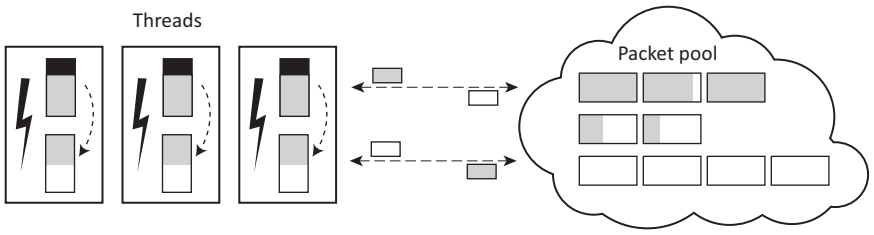


Figure 14.3: Grey packets. Each thread exchanges an empty packet for a packet of references to trace. Marking fills an empty packet with new references to trace; when it is full, the thread exchanges it with the global pool for another empty packet.

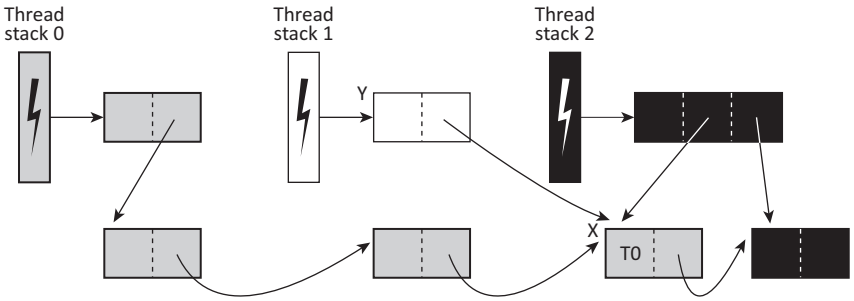
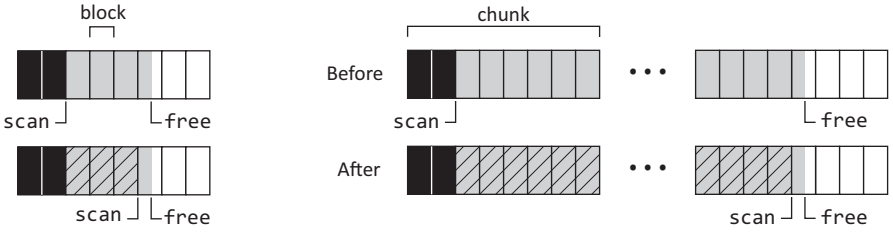


Figure 14.4: Dominant-thread tracing. Threads 0 to 2, coloured grey, white and black, respectively, have traced a graph of objects. Each object is coloured to indicate the processor to which it will be copied. The first field of each object is its header. Thread T0 was the last to lock object X.



(a) Scan pointer and free pointer in the same chunk

(b) Scan pointer and free pointer in different chunks

Figure 14.5: Chunk management in the Imai and Tick [1993] parallel copying collector, showing selection of a scan block before (above) and after (below) overflow. Hatching denotes blocks that have been added to the global pool.

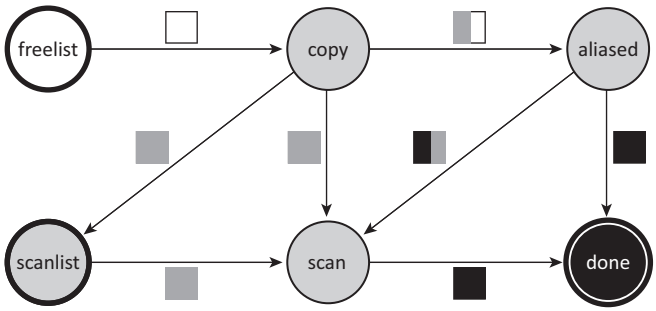


Figure 14.6: Block states and transitions in the Imai and Tick [1993] collector. Blocks in states with thick borders are part of the global pool, those with thin borders are owned by a thread.

copy	scan		
	aliased	▒ or ◧	■
◧	(continue scanning)	(continue scanning)	scan → done copy → aliased
□	aliased → copy scanlist → scan	(continue scanning)	scan → done scanlist → scan
◧	aliased → copy scanlist → scan	(cannot happen)	(cannot happen)
▒	aliased → scan freelist → copy	copy → scanlist freelist → copy	scan → done copy → scan freelist → copy
◧	aliased → scan freelist → copy	(cannot happen)	(cannot happen)
■	aliased → done freelist → copy scanlist → scan	(cannot happen)	(cannot happen)

Table 14.1: State transition logic for the Imai and Tick collector

Figure 14.7: Block states and transitions in the Siegwart and Hirzel collector. Blocks in states with thick borders are part of the global pool, those with thin borders are local to a thread. A thread may retain one block of the *scanlist* in its local cache.

Copyrighted figure withheld

Figure 14.8: State transition logic for the Siegwart and Hirzel collector.

Copyrighted figure withheld

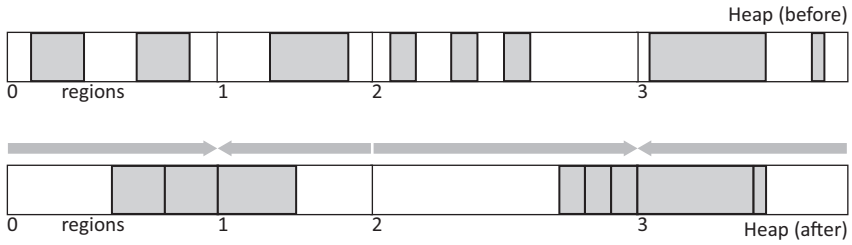


Figure 14.9: Flood *et al.* [2001] divide the heap into one region per thread and alternate the direction in which compacting threads slide live objects (shown in grey)

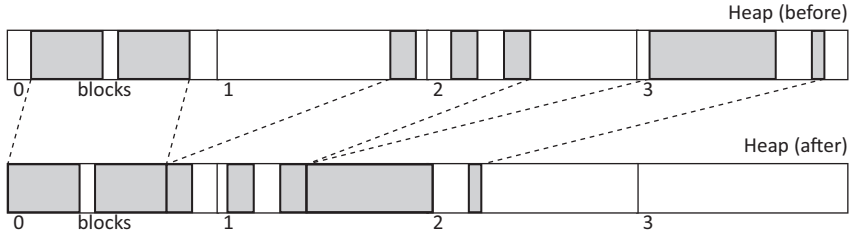
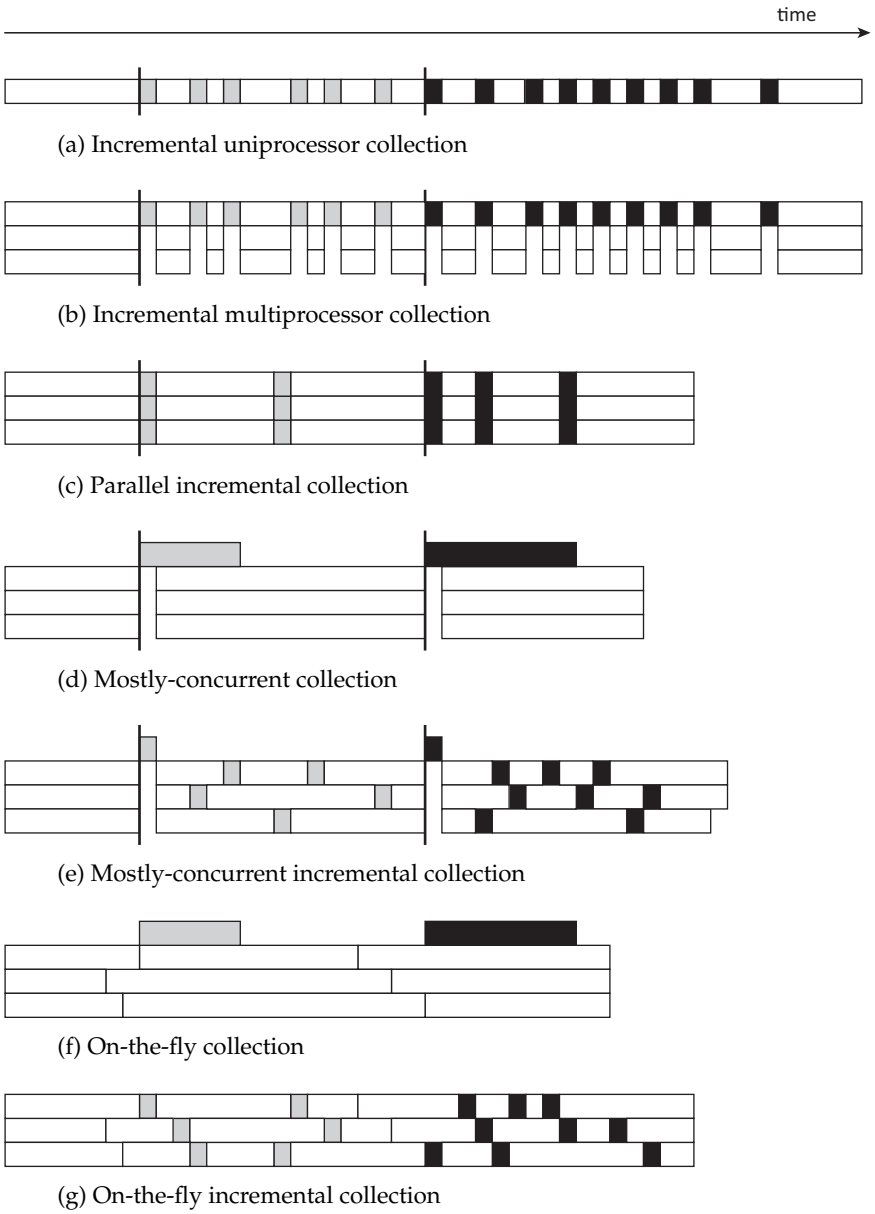


Figure 14.10: Inter-block compaction. Rather than sliding object by object, Abuaiadh *et al.* [2004] slide only complete blocks: free space within each block is not squeezed out.

Figure 14.11: Intel Processor Graphics Gen11, showing a core processor, System-on-a-Chip and its ring interconnect achitecture

Chapter 15

Concurrent garbage collection



(a) Incremental uniprocessor collection

(b) Incremental multiprocessor collection

(c) Parallel incremental collection

(d) Mostly-concurrent collection

(e) Mostly-concurrent incremental collection

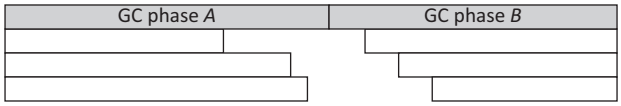
(f) On-the-fly collection

(g) On-the-fly incremental collection

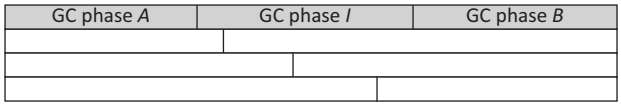
Figure 15.1: Incremental and concurrent garbage collection. Each bar represents an execution on a single processor. The coloured regions represent different garbage collection cycles.

Figure 15.2: The lost object problem: a reachable white object is hidden from the collector by making it unreachable from any grey object.

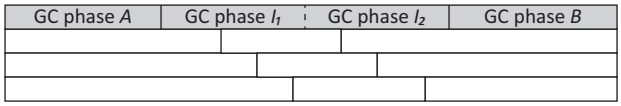
Copyrighted figure withheld



(a) Mostly-concurrent collection



(b) On-the-fly collection, type I

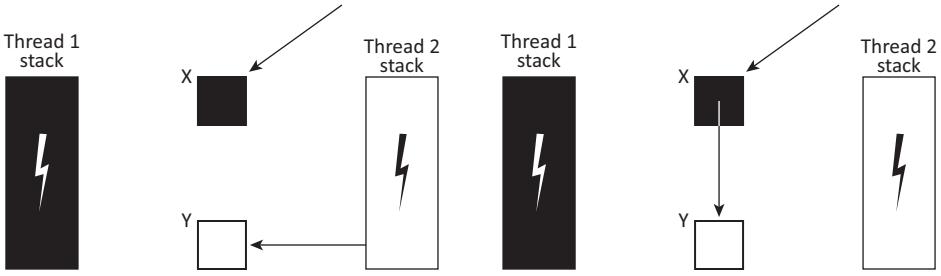


(c) On-the-fly collection, type II

Figure 15.3: Ragged phase changes

Chapter 16

Concurrent mark-sweep



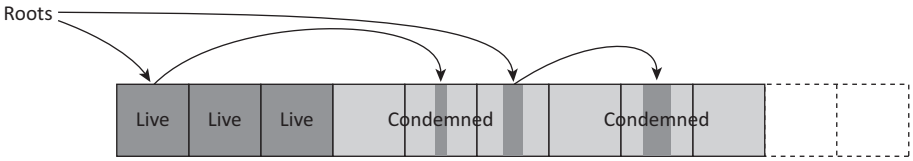
(a) The deletion barrier is 'on'. Thread 1 has been scanned, but thread 2 has not. X has been newly allocated black.

(b) X is updated to point to Y; thread 2's reference to Y is removed. Neither action triggers a deletion barrier.

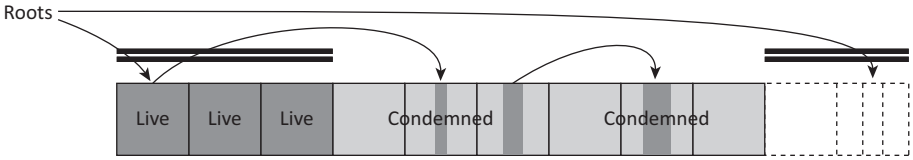
Figure 16.1: On-the-fly collectors that allocate black need more than a deletion barrier to prevent the scenario of a white object reachable only from a black object

Chapter 17

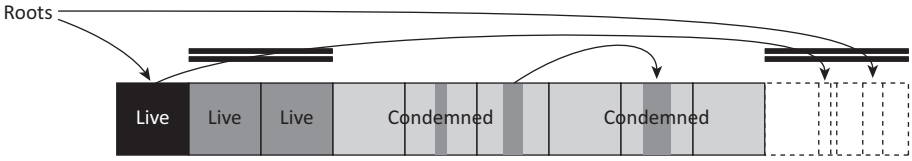
Concurrent copying and compaction



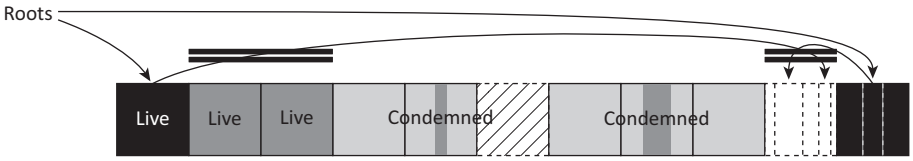
(a) Initial Compressor configuration. All pages are in fromspace.



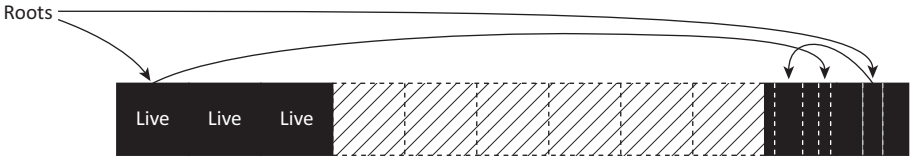
(b) Compute forwarding information, protect all tospace pages (illustrated by the double horizontal bars). These include those reserved to hold evacuated objects and those Live pages not condemned for evacuation. Then flip mutator roots to tospace. Mutators accessing a protected tospace page will now trap.



(c) Trapping on a Live page forwards pointers contained in that page to refer to their tospace targets. Unprotect the Live page once all its stale fromspace references have been replaced with tospace references.



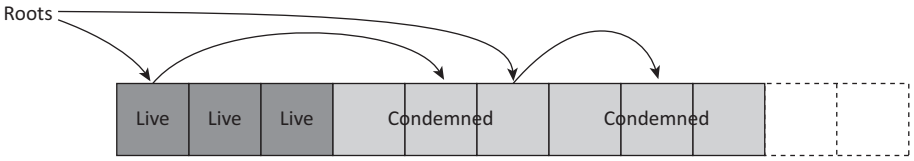
(d) Trapping on a reserved tospace page evacuates objects from fromspace pages to fill the page. The fields of these objects are updated to point to tospace. Unprotect the tospace page and unmap fully evacuated fromspace pages (releasing their physical pages, shown as hatched).



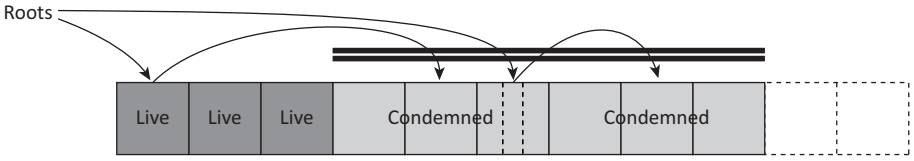
(e) Compaction is complete when all Live pages have been scanned to forward references they contain, and all live objects in condemned pages have been copied into tospace and the references they contain have been forwarded.

Figure 17.1: Compressor

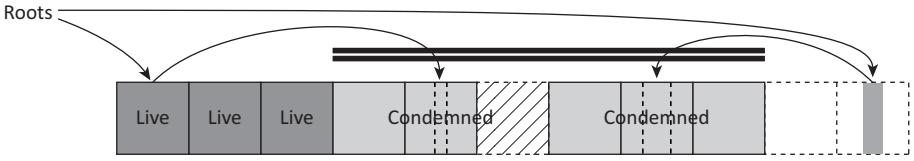
Figure 17.2: C4's tagged pointer layout. The a bits are the address. The two SS bits identify the space (generation) in which the object resides and N is the NMT bit. The virtual address to which the pointer refers is indicated by the p (page number) bits and the a bits (address within the page).



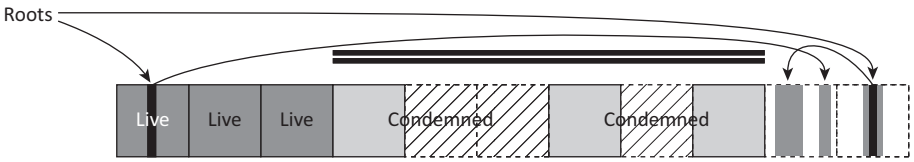
(a) Initial Pauseless configuration. All pages are in fromspace.



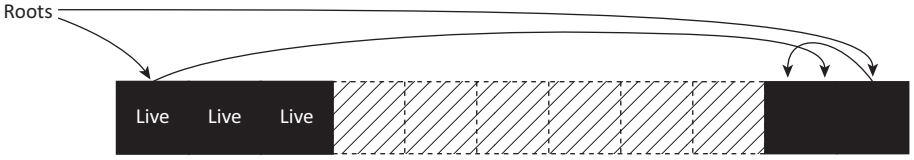
(b) Compute forwarding information, protect all condemned fromspace pages (illustrated by the double horizontal bars), but leave tospace pages unprotected. These include those reserved to hold evacuated objects and those live pages not condemned for evacuation.



(c) Flip mutator roots to tospace, copying their targets, but leaving the references they contain pointing to fromspace. Mutators accessing an object on a protected fromspace page will trap and wait until the object is copied.



(d) Mutators loading a reference to a protected page will now trigger the LVB, copying their targets.



(e) Compaction is finished when all live objects in condemned pages have been copied to tospace, and all tospace pages have been scanned to forward references they contain.

Figure 17.3: Pauseless

(a) Non-generational tagged pointer layout. The F bit is used for concurrent marking through with finalisers, the R bit is the relocated bit, and the two MM bits are mark bits.

(b) Generational tagged pointer layout. The four R bits indicate the good colour (only one of which is set at any time), the M and m bits are mark bits for the old and young generations, respectively, the F bits are used for concurrent marking through with finalisers and the two r bits indicate whether the field is tracked in a remembered set.

Figure 17.4: ZGC tagged pointer layouts. The a bits are the address the pointer holds (with 64-bit alignment, the three lower-order bits are `000`).

Chapter 18

Concurrent reference counting

Thread 1 `Write(o,i,x)`	**Thread 2** `Write(o,i,y)`
`addReference(x)`	`addReference(y)`
`old ←o[i]`	`old ←o[i]`
`deleteReference(old)`	`deleteReference(old)`
`o[i]←x`	`o[i]←y`

Figure 18.1: Reference counting must synchronise the manipulation of counts with pointer updates. Here, two threads race to update an object field. Note that `old` is a local variable of each thread's `Write` method.

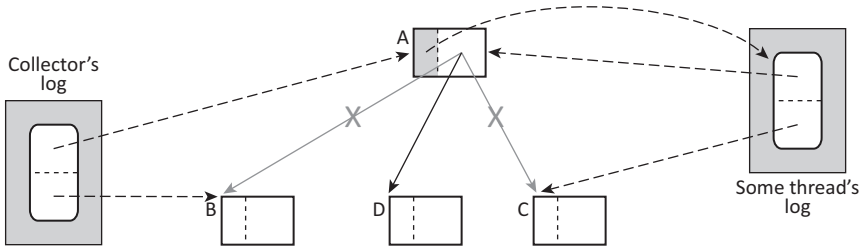


Figure 18.2: Concurrent coalesced reference counting: in the previous epoch A was modified to point to C and the values of its reference fields logged. However, A has been modified again in this epoch (to point to D), and so marked dirty and logged again. The original referent B can be found in the collector's global log, just as in Figure 5.2. The reference to C that was added in the previous epoch will be in some thread's current log: this log can be found from A's `getLogPointer` field.

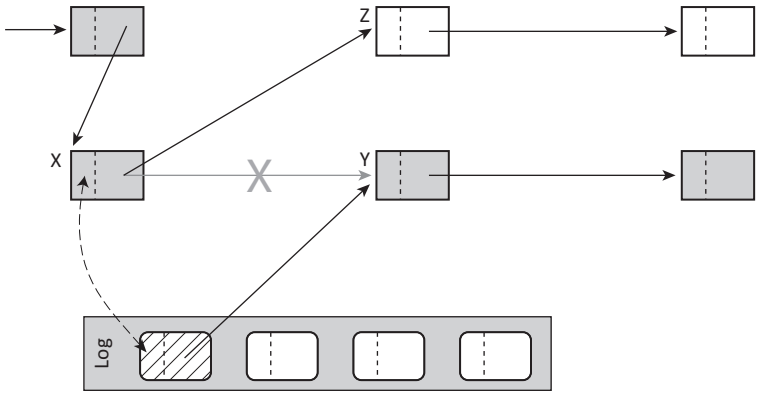


Figure 18.3: Sliding views allow a fixed *snapshot* of the graph to be traced by using values stored in the log. Here, the shaded objects indicate the state of the graph at the time that the pointer from X to Y was overwritten to refer to Z. The old version of the graph can be traced by using the value of X's field stored in the log.

Chapter 19

Real-time garbage collection

Figure 19.1: Unpredictable frequency and duration of conventional collectors. Collector pauses in grey.

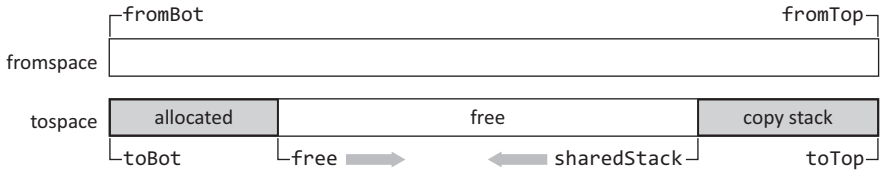


Figure 19.2: Heap structure in the Blelloch and Cheng work-based collector

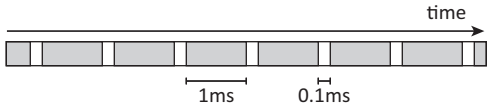


Figure 19.3: Low mutator utilisation even with short collector pauses. The mutator (white) runs infrequently, while the collector (grey) dominates.

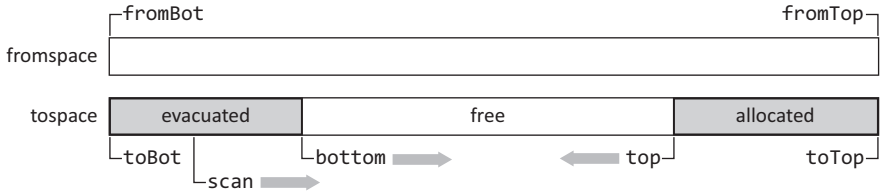


Figure 19.4: Heap structure in the Henriksson slack-based collector

Figure 19.5: Lazy evacuation in the Henriksson slack-based collector

Copyrighted figure withheld

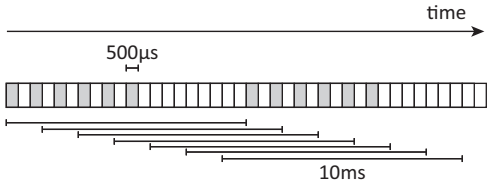


Figure 19.6: Metronome utilisation. Collector quanta are shown in grey and mutator quanta in white.

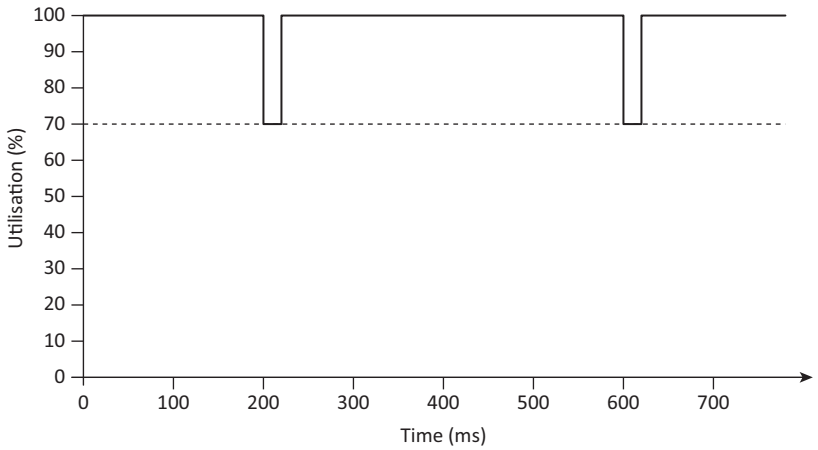


Figure 19.7: Overall mutator utilisation in Metronome

Figure 19.8: Mutator utilisation in Metronome during a collection cycle

Copyrighted figure withheld

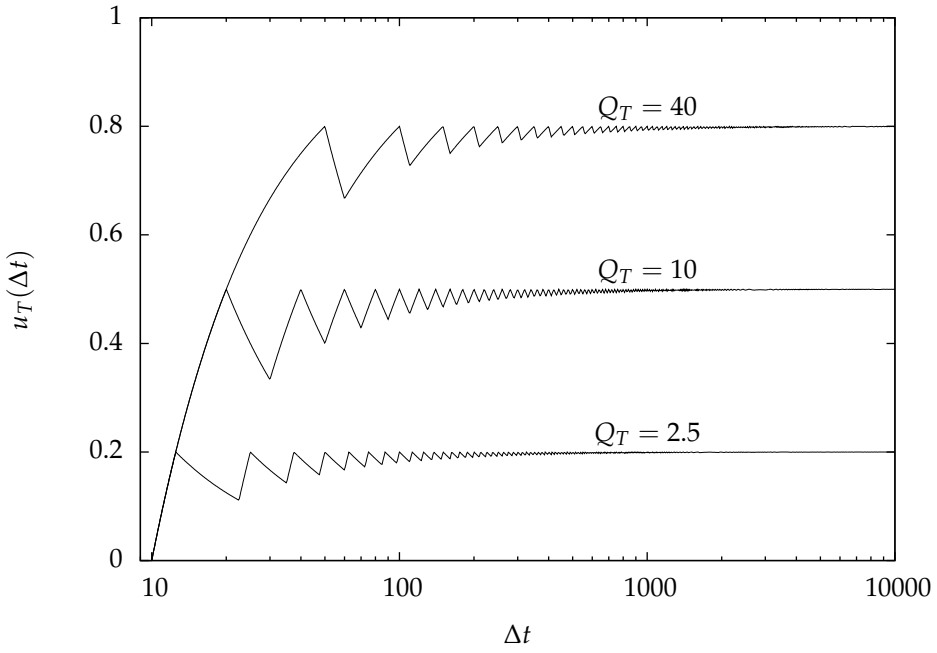


Figure 19.9: Minimum mutator utilisation $u_T(\Delta t)$ for a perfectly scheduled time-based collector. $C_T = 10$. Utilisation converges to $\frac{Q_T}{Q_T+C_T}$. Increasing the frequency of the collector (reducing the mutator quantum) produces faster convergence.

Figure 19.10: Fragmented allocation in Schism

Chapter 20

Energy-aware garbage collection

Chapter 21

Persistence and garbage collection

Bibliography

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. (page 51)

Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989. doi: 10.1002/spe.4380190206. (page 29)

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. (page 33)

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.

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

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. (page 68)

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.

C.J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970. doi: 10.1145/362790.362798. (pages 12 and 13)

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.

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. (pages 48 and 51)

GC 1990, Eric Jul and Niels-Christian Juul, editors. *OOPSLA/ECOOP Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, Canada, October 1990. doi: 10.1145/319016.319042.

GC 1991, Paul R. Wilson and Barry Hayes, editors. *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. doi: 10.1145/143776.143792.

GC 1993, J. Eliot B. Moss, Paul R. Wilson and Benjamin Zorn, editors. *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

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

Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998. https://lucris.lub.lu.se/ws/portalfiles/portal/5860617/630830.pdf. (page 68)

Akira Imai and Evan Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1030–1040, 1993. doi: 10.1109/71.243529. (pages 49 and 50)

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

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

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

ISMM 2004, David F. Bacon and Amer Diwan, editors. *4th ACM SIGPLAN International Symposium on Memory Management*, Vancouver, Canada, October 2004. ACM Press. doi: 10.1145/1029873.

ISMM 2006, Erez Petrank and J. Eliot B. Moss, editors. *5th ACM SIGPLAN International Symposium on Memory Management*, Ottawa, Canada, June 2006. ACM Press. doi: 10.1145/1133956. (page 79)

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

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

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

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

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

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.

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.

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]. (page 77)

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]. (page 34)

JVM 2001. *1st Java Virtual Machine Research and Technology Symposium*, Monterey, CA, April 2001. USENIX Association. https://www.usenix.org/legacy/event/jvm01/. (page 75)

Haim Kermany and Erez Petrank. The Compressor: Concurrent, incremental and parallel compaction. In PLDI 2006, pages 354–363. doi: 10.1145/1133981.1134023. (page 9)

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/. (page 77)

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.

LFP 1984, Guy L. Steele, editor. *ACM Conference on LISP and Functional Programming*, Austin, TX, August 1984. ACM Press. doi: 10.1145/800055. (page 77)

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

David A. Moon. Garbage collection in a large LISP system. In LFP 1984, pages 235–245. doi: 10.1145/800055.802040. (page 14)

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.

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.

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.

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.

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. (page 75)

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.

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.

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.

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.

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.

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.

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. (page 75)

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.

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.

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.

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. (page 77)

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.

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

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

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.

Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Stanford University, 1988. Technical Report CSL-TR-88-351. (page 28)

David Siegwart and Martin Hirzel. Improving locality with parallel hierarchical copying GC. In ISMM 2006, pages 52–63. doi: 10.1145/1133956.1133964. (page 50)

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. (page 34)

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. (page 79)

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