We make the figures from *The Garbage Collection Handbook: The Art of Automatic Memory Management*, Richard Jones, Antony Hosking, Eliot Moss (Chapman and Hall, 2011) 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 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 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
Figure 4.1: Copying garbage collection: an example
A B C D E
L
Fromspace
Tospace
C
L' A' E' B' 
free scan
(d) Scan A’s replica, and so on…

A B C D E
L
Fromspace
Tospace
L' A' E' B' D' C' 
free scan
(e) Scan C’s replica.

A B C D E
L
Fromspace
Tospace
L' A' E' B' C' D' 
free scan
(f) Scan D’s replica. \texttt{scan=free} so collection is complete.

Figure 4.1 (continued): Copying garbage collection: an example
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.

© 2003 Association for Computing Machinery, Inc. Reprinted by permission.
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 \texttt{markGrey}.

(b) After \texttt{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 \texttt{X}, which is still reachable, has a non-zero reference count.

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

\textbf{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.

With kind permission from Springer Science+Business Media: Bacon and Rajan [2001], figure 3, page 214.
Chapter 6

Comparing garbage collectors
Figure 6.1: A simple cycle
Chapter 7

Allocation
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.
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: Intergenerational 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 be not be copied or will be promoted (copied) under different copy count regimes.

© 1989 Association for Computing Machinery, Inc. Reprinted by permission.
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.

© 1989 Association for Computing Machinery, Inc. Reprinted by permission.
(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.
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.

© 2002 Association for Computing Machinery, Inc. Reprinted by permission.
Chapter 10

Other partitioned schemes
Figure 10.1: The Treadmill collector: objects are held on a double-linked list. Each of the four segments hold 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.

(a) Before collecting car 1, train 1 (T1C1).

(b) After collecting car 1, train 1. X moved to the same car as its referent Y, A and B to a fresh train T3. The next collection cycle will isolate T2 and reclaim it wholesale. Numbered labels show the copies made in each algorithm step.

**Figure 10.2**: The Train copying collector.

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.

© 2005 Association for Computing Machinery, Inc. Reprinted by permission.
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.

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 line as implicitly marked: the allocator will not use it.

Blackburn and McKinley [2008], doi: 10.1145/1375581.1375586.
© 2008 Association for Computing Machinery, Inc. Reprinted by permission.
(a) After marking (live objects are shown grey).

(b) After the first copying pass. B has been evacuated and the first block has been unmapped.

(c) After the second copying pass. Note that there was sufficient room to evacuate three blocks.

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.

© 2003 Association for Computing Machinery, Inc. Reprinted by permission.
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.

© 2003 Association for Computing Machinery, Inc. Reprinted by permission.
Chapter 11

Run-time interface
To determine whether a value $p$ is a pointer to an allocated object:

1. Does $p$ point between the lowest and highest plausible heap addresses?

2. Use high order bits of $p$ as an index into the first-level table to obtain the second-level table. In a 64-bit address space, the top-level table is a chained hash table rather than an array.

3. Use middle order bits of $p$ as an index into the second-level table to get the block header.

4. Is the offset of the supposed object a multiple of $\text{hb\_size}$ from the start of the block?

5. Consult the object map for blocks of this size; has the slot corresponding to this object in this block been allocated?

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.

(a) Stack scanning: walking from the top

Figure 11.2: Stack scanning

main()
old IP: ...
saved:
  1: 155
  2: p
  3: 75
locals:
r1 = p
r2 = 784

f()
old IP: main+52
saved:
  1: 784
  2: p
  3: -13
  4: q
locals:
r1 = r
r2 = 17

b()
old IP: f+178
saved:
  1: 17
  2: r
  3: -7
  4: s
locals:
r1 = r
r2 = t

GC happens
IP = g+36
r1 = r
r2 = t
GC happens
IP =
g+36
r1 = r'
r2 = t'
g()
old IP:
f+178
saved:
  1: 17
  2: p'
  3: -7
  4: q'
locals:
pointerRegs: r1, r2
r1 = p'
r2 = 784

# Fig. 11.2 (continued): Stack scanning

(b) Stack scanning: walking back to the top

figure

- Restore calleeSavedRegs
  @ main+52

- Restore
  ⟨r1, r⟩
  ⟨r2, 17⟩

calleeSavedRegs
  ⟨r1, 2⟩, ⟨r2, 1⟩
  @ f+178

- Restore
  ⟨r2, t⟩

calleeSavedRegs
  ⟨r2, 1⟩
  @ g+36

- Restore

- Done
  r1
  r2

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.
Figure 12.4: Restructuring to force finalisation order in cyclic object graphs
Figure 12.5: Finalising in order. Application objects A and B are unreachable from the application and we want to finalise them in that order. Phantom A has a phantom reference to A and a strong reference to B.
Chapter 13

Concurrency preliminaries
Chapter 14

Parallel garbage collection
Figure 14.1: Stop-the-world garbage collection: each bar represents an execution on a single processor. The coloured regions represent different garbage 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 1 to 3, coloured black, grey and white 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.
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.

<table>
<thead>
<tr>
<th>copy</th>
<th>aliased</th>
<th>scanlist → scan</th>
<th>scanlist → scan</th>
<th>scanlist → scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>□</td>
<td>(continue scanning)</td>
<td>(continue scanning)</td>
<td>(cannot happen)</td>
<td>(cannot happen)</td>
</tr>
<tr>
<td>□</td>
<td>aliased → copy</td>
<td>(continue scanning)</td>
<td>(cannot happen)</td>
<td>(cannot happen)</td>
</tr>
<tr>
<td>□</td>
<td>aliased → copy</td>
<td>(cannot happen)</td>
<td>(cannot happen)</td>
<td>(cannot happen)</td>
</tr>
<tr>
<td>□</td>
<td>aliased → scan</td>
<td>copy → scanlist</td>
<td>(cannot happen)</td>
<td>(cannot happen)</td>
</tr>
<tr>
<td>□</td>
<td>aliased → done</td>
<td>(cannot happen)</td>
<td>(cannot happen)</td>
<td>(cannot happen)</td>
</tr>
</tbody>
</table>

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.

© 2006 Association for Computing Machinery, Inc. Reprinted by permission.

Table 14.2: State transition logic for the Siegwart and Hirzel collector.

© 2006 Association for Computing Machinery, Inc. Reprinted by permission.
Figure 14.8: 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 14.9: Inter-block compaction. Rather than sliding object by object, Abuaiaadh et al [2004] slide only complete blocks: free space within each block is not squeezed out.
Chapter 15

Concurrent garbage 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.

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 & 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 finished 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
(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 a GC-trap via the read barrier, copying their targets.

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

**Figure 17.2:** Pauseless
Chapter 18

Concurrent reference counting
<table>
<thead>
<tr>
<th>Thread 1 Write(o,i,x)</th>
<th>Thread 2 Write(o,i,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addReference(x)</td>
<td>addReference(y)</td>
</tr>
<tr>
<td>old ← o[i]</td>
<td>old ← o[i]</td>
</tr>
<tr>
<td>deleteReference(old)</td>
<td>deleteReference(old)</td>
</tr>
<tr>
<td>o[i]←x</td>
<td>o[i]←y</td>
</tr>
</tbody>
</table>

**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.
Some thread’s log

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
(a) Before a high-priority task performs $B.y \leftarrow A.x$. The write barrier catches the assignment since the fromspace $C$ object is not previously evacuated or scheduled for evacuation.

(b) After having reserved a tospace location for $C$. A temporary \texttt{toAddress} pointer (dashed) to the reserved area prevents multiple tospace reservations for $C$. Forwarding pointers prevent access to the uninitialised reserved space.

(c) When the high-priority task pauses, the collector finishes evacuating $C$ to its reserved tospace location, and sets the forwarding pointers to refer to the tospace copy. $A.x$ will be forwarded later when the $A$ object is scanned by the collector.

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

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.
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.
(a) A two-fragment object with a payload of from six to twelve words. The sentinel fragment has three header words: a fragmentation pointer to the next object fragment, a garbage collection header and a type header. Each fragment has a header pointing to the next.

(b) A single-fragment array with a payload of up to four words. The sentinel fragment has four header words: a null fragmentation pointer, a garbage collection header, a type header and an actual length \( n \leq 4 \) words, followed by the inlined array fields.

(c) A multi-fragment array with a payload of up to three fragments (up to 24 words). The sentinel fragment has five header words: a non-null fragmentation pointer to the inlined array spine, a garbage collection header, a type header, a pseudo-length \( 0 \) indicating fragmentation and an actual length \( 4 < n \leq 24 \) words (at the same negative offset from the spine as in (b)), followed by the inlined spine. Payload fragments have no headers.

(d) An array with a payload of four or more fragments (more than 24 words). The sentinel fragment has four header words: a non-null fragmentation pointer to the separately allocated array spine, a garbage collection header, a type header and a pseudo-length \( 0 \) indicating fragmentation, followed by the rest of the sentinel which is unused. The spine has a two-word header: the actual length and a forwarding pointer at negative offsets. Payload fragments have no headers.

Figure 19.10: Fragmented allocation in Schism.

Pizlo et al [2010b], doi: 10.1145/1806596.1806615.
© 2010 Association for Computing Machinery, Inc. Reprinted by permission.
Bibliography

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


Joel F. Bartlett. SCHEME->C: a portable Scheme-to-C compiler. WRL Research Report 89/1, DEC Western Research Laboratory, Palo Alto, CA, January 1989b.


Hans-Juergen Boehm. Mark-sweep vs. copying collection and asymptotic complexity. 


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


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


Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In GC 1993.


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. This report summarises King [2004].


Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal, 30(3), March 2005.


Colophon

This book was set in Palatino (algorithms in Courier) with pdftex (from the TeX Live 2010 distribution). The Illustrations were drawn with Adobe Illustrator CS3. We found the following packages to be useful: comment (comments), paralist (in-paragraph lists), geometry (page size), crop (crop marks), xspace (space suppression), setspace (line spacing), fnbreak (detect footnotes spread over more than one page), afterpage (page break control), multicol (multiple columns), tabularx (tabular material), multirow (multiple rows and columns in tables), dcolumn (“decimal points” in tables), graphicx and epstopdf (graphics), subfig (subfigures), rotating (rotate objects), listings (algorithm listings), caption (captions), mathpazo (typesetting mathematics to match Palatino body text), amssymb and amsmath (mathematics), hyperref (hyperlinks and back-references), glossaries (glossaries), natbib (bibliography), makeidx and index (index).