Dynamic memory management: challenges for today and tomorrow

Richard Jones
Computing Laboratory, University of Kent
R.E.Jones@kent.ac.uk

Abstract

Garbage collection is a key component of almost all modern programming languages. The advent of conventional object-oriented languages supported by managed run-times (e.g. Java, C♯ and even Managed C++) has brought garbage collection into the mainstream and, as memory manager performance is critical for many large applications, brought it to the attention of programmers outside its traditional functional programming language community.

In this paper, I review how garbage collection got to where it is today, why it is desirable, what performance you might reasonably expect and I shall outline the directions in which research is moving. In particular, I look at some of the challenges facing modern garbage collection, in contexts ranging from collection for high-performance, multiprocessor systems to collection for real-time systems, from better integrating with its operating environment to supporting specific applications. I speculate on future directions for research.

1 Automatic or explicit memory management?

1.1 Why garbage collect?

To the functional programmer, the answer is obvious and it may even seem that the question is not worth asking. In a lazy functional programming language, sharing and delayed execution of suspensions means that execution order is hard to predict: it is extremely difficult to determine the point at which a heap object is no longer used. Even if an automatic technique (such as region inference [31] is used) a garbage collector is often required as a backup in order to run space-efficiently. In any case, for many languages, the simplest answer is that garbage collection is needed because it is mandated by the language specification.

But this response is too defensive. It invites the criticism that automatic memory management is only a requirement because of the 'deficiencies' of the language (for example, Java does not have C++'s automatic objects, constructors and destructors [61]), or that '[insert your favourite language] programmers are lazy'. While there is some anecdotal evidence (certainly from scrutiny of student projects) that, say, Java programmers often do not take as much care managing memory as they might, there are powerful reasons for advocating automatic memory management, independent of choice of language.

Explicit memory management is antagonistic to two of the most powerful tools for the management of complexity in large-scale software systems are *modularity* and *abstraction*. In order to be reused in different contexts, software modules should have interfaces that are simple and well-defined. However, 'liveness is a *global* property' [67]: adding memory management book-keeping detail to interfaces weakens the abstractions and reduces extensibility of modules. Worse, changes may radiate beyond the module being developed — for example, space leak or premature reclamation in one module might lead to the failure of another. Garbage collection, on the other hand, uncouples the problem of memory management from class interfaces, rather than dispersing it throughout the code. This is why it has been a fundamental component of many object-oriented languages.

1.2 The cost of garbage collection

Nevertheless, automatic memory management does impose a performance penalty on the program, but not as much as is commonly assumed. Remember that explicit operations like malloc/free also impose a significant cost. Hertz and Berger measured the true cost of garbage collection for a variety of Java benchmarks and collection algorithms [33]; King and Jones compared the costs of conservative mark-sweep against smart-pointer reference counting and the Windows allocator.

Hertz and Berger instrumented the Jikes RVM Java virtual machine [3, 2] to discover precisely when objects became unreachable. They used the reachability trace as an oracle to drive a simulator [35], measuring cycles and cache misses. For the garbage collector configurations, they used a number of different garbage collectors, including simple semi-space copying and mark-sweep collectors, two generational collectors and two hybrid collectors. For the explicit configurations, the simulator invoked `free` at the point where the trace indicated that an object had become garbage, using the Lea [40] allocator (version 2.7.2). Note that the simulator only measured true costs, i.e. that of the program being run, Jikes RVM and either the collector or the malloc/free implementation; the oracle cost nothing. Although, as expected, results varied between both collectors and explicit allocators, Hertz and Berger found that a generational collector, with a variable sized (Appel-style) nursery and a mark-sweep old generation, matched the execution time performance of explicit allocation when given a heap $5\times$ the minimum required. If the heap size was reduced to $3\times$, the overhead increased to 17% on average.

King and Jones compared the performance of Boehm's conservative collector, version 6.2 [13], Boost's 'shared' and 'intrusive' reference counter implementations [1], Lea's allocator (again version 2.7.2), and the standard allocator provided by Microsoft VisualStudio.NET vc7.0 compiler. The testbed was a version of the gcbench[2] tree-manipulation program, with different depths of tree and numbers of threads. The system managed by the conservative collector had an elapsed time was just 27% of that managed by the Windows allocator, also outperforming both reference counters by a substantial margin. The Lea allocator was fastest in all cases. However, all these allocators used more memory than the Windows allocator: Lea 125%, Boehm 150% and Boost 152–166%.

[1]http://www.boost.org

[2]http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench

2 The Lisp legacy

Lisp has been the birth place of too many garbage collection technologies to list in full here, but all the techniques italicised below are significant Lisp 'firsts'. *Mark-Sweep* [43], invented by McCarthy in 1958, remains today the dominant technique for managing the oldest generation in generational collectors and for incremental and concurrent collection. *Reference counting* (1960) [20] offers the benefit that costs are distributed throughout the computation, thus avoiding pauses incurred by stop-the-world tracing collectors. However, reference counting local and temporary variables inflicts too much overhead though incrementing and decrementing reference counts. *Deferred reference counting* (1976) [22] removes this overhead by delaying examination of local variables. Despite its attractions, reference counting remained largely out of favour for many years because of its inability to reclaim cyclic data and the cost of reference counting operations, particularly for threads. However, as we shall see later, recent techniques have led to a revival of interest.

Semi-space copying [27, 15] was first suggested in 1963 by Minsky who proposed copying the heap to disk. The Fenichel & Yochelson (1969) collector used an explicit stack for copying, but Cheney's elegant solution (1970) embeds the stack into toSpace data, thereby requiring no extra space (although a number of modern collectors have reverted to using auxiliary data structures for better management of work when copying with parallel threads). The *Treadmill* [8] is a non-moving collector that shares the advantages of copying collection (i.e. complexity proportional to live data); it is used today, for example, for management of 'large object spaces', in which objects do not share pages and hence external fragmentation is not an issue [9].

Copying collection lies at the heart of modern, region-based collection. The most widely used form of region-based collection is *generational collection* (1978) [42]. Generational collectors exploit the observation that 'most objects die young' [63] by segregating objects into regions ('generations') by age and collect younger generations more frequently than older ones. They must be able to identify cross-region pointers, and this is usually done by intercepting pointer writes. Usually, such write barriers are implemented in software, but *hardware barriers* have also been used [42, 44, 7]. One of the most widely used structures for recording the location of such pointers is the *card table* (1988) [58]; the Symbolics 3600 used a similar page table [44]. Both copying collection and *Mark-Compact collection* (1964) [32, 26], which moves or, better, slides live (marked) data to one end of the heap, remove fragmentation thus allowing fast, linear ('bump a pointer') allocation.

The first *parallel garbage collection* algorithm was due to Steele (1975) [59]. Like Dijkstra's better known algorithm [23], it used an *incremental update write barrier*. Although IU write barriers tend to reduce the volume of floating garbage compared to other techniques, they require a final, stop the world phase to terminate the marking phase in order to detect any pointers that the mutator might have inserted into, for example, stack locations already scanned. Yuasa's *snapshot at the beginning barrier* (1990) [70], developed for Kyoto Common Lisp, although permitting more floating garbage than an IU barrier, allows termination without this final stop the world phase. It is surprising that, given this advantage, SAB barriers have only recently been adopted for high performance, concurrent collectors.

Non-moving collectors suffer from fragmentation, but compacting the heap in a concurrent/incremental context is challenging as all references to a moved object must be updated. The best known *incremental copying collector* is Baker's (1978) [7]. In order to ensure that the mutator sees only toSpace objects, Baker uses a *read barrier*. Read barriers have long been considered prohibitively expensive compared with write barriers on conventional hardware (as pointer reads are more common than writes). However, recent studies have suggested that well-engineered read barriers may not be as expensive as feared [10, 72]. Compaction phases of mark-compact collectors are expensive, typically taking many times longer than the marking phase. One solution is to *compact the heap incrementally*, by dividing it into a number of segments, each of which is compacted separately, either by treating two of the segments (one of which should be empty) as semi-spaces, or simply by compacting a single segment in place [39]. In this way, mutator threads need to be stopped only long enough to compact a single segment.

Locality is vital for performance, especially in modern deeply pipelined architectures. Certainly, paging is inimical to good performance, but good cache behaviour is also important. White suggested (1980) *using the collector to improve locality* [66]. Moon (1984) modified Cheney's algorithm to *copy objects approximately breadth-first*. Others have had the collector reorganise 'hot' fields of objects. The first studies of garbage collector cache behaviour were by Zorn (1989) [73].

Perhaps some of the most interesting garbage collection development work was done on Lisp machines[3] in the mid-80s. These architectures did not endure because they could not compete commercially against stock hardware. Although garbage collection research and development has continued in the Lisp community[4], nothing has been published recently. To the best of my knowledge, the last garbage collection paper from the Lisp community was published in 2000 [57]. This is a shame. It is important that the lessons learnt in the past do not become lost to the wider community.

3 The future

Despite almost 40 years of garbage collection research, new challenges continue to emerge. As we have

[3] I use the term generically to include Lisp Machines, Symbolics, Explorer...

[4] For example in Harlequin's (now Ravenbrook's) MPS system.

seen above, modern collectors can offer throughput competitive with that of the best malloc implementations. Pause times are generally not intrusive. On a 2.53GHz Pentium 4, 512KB cache, 512MB RAM, running Debian Linux, kernel 2.6.8, average pause times for SpecJVM98 benchmarks under the HotSpot Client VM (Blackdown 1.4.1) were 1.6ms (minor collection) and 22ms (full collection), with standard deviations 1.7 and 15.0 respectively.

However, these are middle-sized benchmarks running on a middle of the range machine. The new challenges lie elsewhere. First, modern servers are heavily multiprocessor. At the top end, the Azul Systems 3840 has 16 processors, each of which has 24 cores. Intel's Polaris architecture, although not yet a general purpose processor, has 80 cores. And dual-core processors, on laptops and desktops, are ubiquitous. Setting aside the question of how we are going to program (correctly) applications to take full advantage of massive parallelism, how will we construct automatic memory management systems that best perform on such systems? Applications running on future servers will have prodigious demands for memory; already heaps of tens of gigabytes are not uncommon. Clearly, stopping the world to collect a huge heap is impractical, but certain operations (for example, updating all references to a moved object) must appear atomic.

At the other end of the scale, the sophistication of embedded devices is increasing. Applications written in 'managed code' running on a virtual machine are attractive for many reasons (safety, portability, ease of update, etc.). These environments provide different challenges. Energy efficiency is important for battery powered devices; there are opportunities here for memory management research (for example, selective powering down of memory banks) but the field is immature. In what other ways may the collector adapt better to its environment? For example, how can the memory management system cohabit better with the operating system (for example, to reduce page faults).

Systems may also demand guaranteed, hard real-time response. Although garbage collection papers have had titles that include the word 'real-time' since 1978, for most of these the term simple means incremental collection without hard guarantees. One solution has been to side-step garbage collection altogether: the Real-Time Specification for Java provides the programmer with memory regions that obey a stack discipline (inter alia). Here all the memory in a region can be reclaimed in a constant-time operation. However, the effect of RTSJ is pervasive, requiring libraries to be rewritten. Can garbage collection be truly real-time?

4 Concurrency and parallelism

4.1 Definitions

Throughout this paper we distinguish serial and parallel; stop the world, incremental, concurrent and on the fly collection. A *serial collector* will employ just one collection thread no matter how many CPUs are available; clearly, this does not make good use of resources. In contrast, a *parallel collector* will share collection tasks between many CPUs; the user program (the *mutators*) may or may not be stopped for garbage collection. *Stop the world collectors* require all mutator threads to be halted for collection. *Incremental collectors* interleave small bits of collection work with the mutator; for example, each allocation request may cause some objects to be marked. *Concurrent collectors* allow mutator threads to run simultaneously with collector threads; these threads may be executed on a single CPU or in parallel on many CPUs. Many concurrent schemes require all mutator threads to be stopped briefly (for example, at the start and end of each phase [28, 48]), others stop mutator threads *on the fly*, just one at a time [41, 50].

4.2 Requirements

Let us suppose we wish to build a garbage collector for a high-performance server system. Such a system will be multiprocessor; application programs will have very large (multi-gigabyte) heaps and will run a large number of threads (many more threads than processors). In general, such a system will require high throughput and short response times, but will not specify hard real-time guarantees. Long pauses, even if infrequent (for example, to compact the heap), are particularly undesirable for servers that provide transaction processing. Any delay in processing work will cause a queue of transactions waiting to be processed to grow. As these transactions time-out, further transaction requests will be submitted, leading to yet more work.

A high-performance system must therefore extract as much parallelism as possible. As far as garbage collection is concerned, the bottlenecks are heap contention when allocating, contention for the mark stack when marking, and handling compaction (moving objects, fixing up addresses). As in any parallel system, it is important that data structures should be lock-free wherever possible, and load balancing is critical (treading the path between work starvation and excessive synchronisation). The general strategy is to over-partition work and share tasks between threads. Most garbage collection work can be partitioned into separate tasks: marking, scanning card tables or remsets, sweeping and compaction.

In the next sections, we shall consider a heap at least partly managed by a high performance mark-sweep collector. Allocation must be fast. It may be a generational collector, in which the young generation will almost certainly be managed by a copying collector. As fragmentation is an issue for all large, long-running systems, we shall require (at least occasional) compaction. In summary, we want concurrent allocation, parallel marker and collector threads, concurrent marking, concurrent sweeping, parallel (and maybe concurrent) compaction.

4.3 Concurrent allocation

Fast allocation is essential for any high performance system. Although the heap is a shared resource, it is essential to avoid any contention, such as locks or atomic instructions (such as CompareAndSwap). The fastest allocation mechanism is simply to compare a free-space pointer against the end of the heap region and increment it by the size of the allocation request if there is room. Clearly, this pointer cannot be shared between threads. In principle, the solution is simple: divide the heap region into a large number of *thread local allocation buffers* many more than the number of mutator threads). Now each thread can allocate without contention within its buffer, and must only compete for a fresh buffer (e.g. with CAS). Garthwaite et al. give good overview of the subtleties required [29]. Locality properties make this mechanism effective even for memory managers that rarely move objects; here, the trick is to allow variable-sized local allocation buffers [24].

4.4 Concurrent marking

Tracing live data concurrently (whether truly concurrently or incrementally) is notoriously tricky [37]. In general, the smaller each block of instructions that must be executed atomically, the more opportunity for parallelism there is. However, it is difficult to relax atomicity constraints on instructions without compromising correctness. Vechev et al. have suggested deriving efficient concurrent algorithms from an 'obviously correct' one through a series of correctness-preserving transformations [64]. But what is an obviously correct algorithm?

4.4.1 Coherency

Asynchronous execution of mutators and collectors introduces a coherency problem. Interleavings are possible in which the mutator 'hides' a live (i.e. reachable) object from the collector. For example, suppose an object holds a single pointer to a child object, and the collector marks the object. If the mutator then copies the child pointer to another, already marked object, and deletes the original pointer before the collector sees the pointer, then the child will never be marked, and so will be reclaimed incorrectly. The tricolour abstraction [23] is useful here. Each object has a colour:

Black: The object and its immediate descendants have been visited (e.g. marked); the marker has finished with this object and need not revisit it.

Grey: The object has been visited, but its components may not have. Alternatively, in incremental or concurrent collection, the mutator has rearranged the connectivity of the graph. In either case, the object must be revisited.

White: The object has not been visited.

Tracing terminates when no grey objects remain: all white objects are garbage. There are two ways to prevent the mutator from interfering with a collection, i.e. by writing white pointers into black objects:

Figure 1: An incremental update barrier, Dijkstra-style (left) and a snapshot at the beginning write-barrier (right), as a pointer is updated.

- Record where the mutator writes black-white pointers, so that the collector can (re)visit objects. Protect objects with a *write barrier*.
- Ensure that the mutator never sees a white object. Whenever the mutator attempts to access a white object, it is visited by the collector. White objects are protected by a *read-barrier*.

Pirinen provides an excellent comparison of barrier strategies [52].

4.4.2 Write barriers

To reclaim an object falsely, two conditions must hold:

1. a pointer to the white object is written into a black object,
2. the original path to the white object is destroyed.

If (1) does not hold, there will be at least one path to each reachable white object that passes through a grey object. If (2) does not hold, the white object will still be reachable through the original reference.

There are two forms of write barrier method (Figure 1): *incremental update* (IU) methods catch changes to connectivity, *snapshot-at-the-beginning* (SAB) methods prevent loss of the original path. The IU barrier traps attempts to install a pointer to a white object into a black object by shading (colouring a white object grey) either the target (Dijkstra-style) or the source object (Steele-style); no special action is required when a pointer is deleted. IU barriers incrementally record changes to the shape of the graph in order to prevent condition (1) arising. SAB barriers shade the *old* target (cf. copy-on-write), preventing condition (2) arising. SAB collectors can be expected to be more conservative than IU collectors since any objects that become unreachable since the start of a collection are preserved. However, they have one important advantage: simpler termination. If no barriers are imposed on writes to local variables, IU collectors require a stop the world phase in which they can scan their roots and complete tracing live data. SAB collectors do not require a final, stop the world phase.

4.4.3 Read barriers

Alternatively, we can prevent the mutator seeing white objects so that it cannot disrupt the collector. Approaches differ on whether the mutator is allowed to see grey objects or not. The best-known read-barrier collector is Baker's semi-space copying collector [7]. Baker permits access to grey, i.e. toSpace, objects but accesses to fromSpace objects are trapped and the object is copied to toSpace.

4.5 Parallel marking

If multiple, parallel markers are used, the mark-stack becomes the focus of contention. The solution is to over-partition marking into more tasks than marker threads. Sun Microsystems' parallel collector reduces contention through *work-stealing* [28]. Each marking thread is given its own mark stack onto the bottom of which it pushes and pops addresses. If a thread exhausts its stack, it steals work from the top of another's stack. Marking terminates when all threads' stacks are empty. Thomas devised *grey packets* for Insignia Solutions' Jeode JVM (an idea reinvented in [48]). Here, each marker thread has an in-packet and out-packet of work (objects to mark). It obtains its (usually full) in-packet from a shared pool. The thread then removes (references to) objects from the packet, marks them, and adds their children (objects which need to be marked) to its out-packet. When the in-packet is empty, it obtains a fresh one from the pool; when an out-packet is full, it returns it to the pool and obtains a fresh, empty packet. Marking terminates when all packets are empty. This technique avoids most contention and allows simple termination. Because the marker does not use a traditional LIFO stack, it also facilitates prefetching data to be marked (thereby avoiding cache stalls). Weak ordering problems are also reduced (fence instructions are only required around packet acquisitions and disposals).

4.6 Sweeping

Sweeping concurrently is comparatively simple since, by definition, the sweeper deals only with dead objects (so cannot interfere with either mutators or markers from the next collection cycle) or GC words in live object headers (which mutators have no access to). The heap can be partitioned into segments for each parallel thread to sweep.

4.7 Compaction

Without compaction, heaps tend to fragment over time, reducing allocation performance. A key problem for compaction is that objects need to be moved, and all references to them updated, in a way that appears to be atomic.

An idea originally due to Lang & Dupont [39] is to divide the heap into a number of regions and compact each separately. The benefit here is that each can be compacted sufficiently fast to meet pause-time requirements. References between regions can be handled by remembered sets (or ‘remsets’), one per region, constructed by the marker threads: each remset holds the location of references into its region. If the heap is to be compacted incrementally, i.e. one region at a time, the condemned region can be determined by an appropriate heuristic (such as the volume or number of live objects it contains, the size of its remset, etc.). The condemned region can then be compacted, using the remsets to fix up references, either in place or by copying to an empty region.

Abuaiadh et al. [1] split the heap into many small *blocks* (e.g. 256 bytes) and a smaller number of large *target areas*, e.g. 16 $\times$ the number of processors. Each compacting thread claims a target to compact and a target into which to move compacted data; both claims are made by incrementing an index (a CAS). Blocks are then copied atomically from the source target to the destination target. Their system reduced compaction time by a factor of 4 for a large three-tier application suffering fragmentation problems. Locality effects meant that moving non-empty blocks en masse, rather than individual object at a time, reduced compaction time by a further 25% at a small cost in space (+4%). Compaction speed-up was linear in number of threads, and the benefit of a compacted heap improved throughput slightly.

The *Garbage-First* collector [21] aims to meet a soft-real time goal as far as possible: garbage collection should use no more than *x*ms in any *y*ms time slice. Again the heap is divided into small, equal-sized regions, protected by a bi-directional, card table based write-barrier. Any write to a clean card causes the card to be dirtied and a pointer to the card to be added to the thread’s small, local remset log; when the log is filled, it is added to a global log. A background thread waits for the global log to become sufficiently large before filtering the log to each region’s remset; processing of ‘hot’ cards is postponed. The world is stopped for the parallel evacuation phase, in which a set of regions to be evacuated is chosen and collector threads compete for work. The collector can be made generational by always condemning allocation regions. Marking can be performed concurrently with compaction by using a current and a previous bitmap and deeming any object above a high water mark (i.e. allocated since marking started) as implicitly live. Compared with the HotSpot ‘low pause time’ collector (parallel young generation copying and concurrent mark-sweep), Garbage-First leads to fewer and less severe violations of soft real-time goals, and scales well with the number of processors.

A number of compaction schemes have been devised that rely on memory protection support. Instead of trapping accesses and moving objects individually, page protection mechanisms can be used. The first collector to do so was the Appel-Ellis-Li collector [4]. This protects grey pages and, on any access violation, the trap handler scans the whole page on which the object resides, copying all objects’ referents (to pages which are then protected) before unprotecting the page and resuming the mutator.

The IBM mostly concurrent collector [49] maps the whole heap into two virtual address ranges. As with the schemes above, the heap is divided into a number of logical regions. Following the mark and sweep phases, a region is chosen for compaction. In a brief stop the world phase, the objects in this region are moved, and roots (addresses in thread stacks, registers, etc) are updated; the whole heap is read/write protected before the mutators are resumed. Other addresses are fixed up either concurrently as the mutators trigger access violations, or by background fix-up threads. Fix-up is done using the second virtual address region to avoid further violations. Separating object relocation from address fix-up reduced compaction pause times to 10-15% of mark time (rather than 10× mark time). A throughput improvement of up to 10% was obtained.

The *Compressor* [38] extends these ideas to reduce the cost of sweeping over the heap to fix-up addresses. An offset table is constructed from the mark-bit vector constructed in the mark phase, and used to compute relocation addresses. Each compaction thread finds a page to compact and moves objects using the offset table; it then traverses these object to update the pointers they contain (again using the offset table) before releasing the from-page. The same page protection strategy is used for concurrent compaction, but this time threads are stopped only to fix-up the roots.

Azul Systems' *Pauseless GC* [19] is designed to operate in an environment of hundreds of threads, 10–100GB heaps on Azul's multicore, multiprocessor hardware. Their custom hardware and operating system provides fast (typically 1 cycle), user-mode trap handling, a garbage collector mode (between user and kernel mode) and large, 1MB pages. Their current implementation, based on the HotSpot JVM (collection safepoints, cooperative preemption), aims for a soft real-time goal of pause times in the range 10–100ms, but high mutator utilisation rate. The collector operates in three phases: mark, relocate, re-map; marking in one epoch operates concurrently with the relocate and re-map phases of the previous one. Each object has two mark-bits, one for this cycle and one for the previous. Marking is parallel and concurrent, using work-lists. Running threads mark their own roots; the roots of blocked threads are marked by marker threads. Each reference has a *Not-Marked-Through* bit, stealing one bit from the 64-bit address space. If a thread loads a reference with its NMT bit set to the wrong value, the mutator gets a trap. The trap handler marks the target and corrects the NMT bit in the reference using a CAS. There is no stop the world phase at the end of marking. In the relocation phase, pages are protected and objects concurrently relocated, either by the collector or by the mutator if it attempts to access a protected page. Forwarding pointers are held in a side-table so *physical*, but not virtual, pages can be recycled as soon as their objects have been evacuated: this means that some references are stale. In the re-map phase, the collector traverses the object graph, tripping the read barriers. Although the Pauseless collector may encounter 'trap storms' at phase boundaries, these are apparently brief.

5 Reference counting revisited

The prime attractions of reference counting are its distribution of memory management costs throughout the computation and its relatively good locality (any pointer write affects the reference counts of only the old and new targets). However, it also has disadvantages. Standard reference counting cannot reclaim cyclic garbage (and such structures are common), so a back-up tracing collector may be necessary. Reference count manipulations must be atomic, which is problematic/expensive for multi-threaded programs. Consequently, reference counting has been largely ignored for many years. However, recently new techniques have been developed that make reference counting attractive once more.

Reference counting is best suited to long-lived data structures that are rarely updated living in huge heaps, whilst short-lived objects with frequently update fields are managed well by copying collectors. *Ulterior reference counting* [12] seeks the benefit of both approaches in a generational collector that manages the young generation in the usual way, but the old generation with reference counting (using trial deletion to handle cyclic structures [5]). In a sense, this is a generalisation of deferred reference counting [22]. In tightly constrained heaps, the collector gave throughput comparable to a fixed-sized nursery, generational mark-sweep collector, but with maximum pause times reduced by a factor of 4.

Levanoni & Petrank [41] observe that, in any sequence of updates to a slot holding a pointer, only the first and last are significant for reference counting. Suppose the slot takes values o_1, o_2,... o_n between garbage collection cycles. A naive implementation would perform `RC(`o_1`)--`, `RC(`$o2$`)++`, `RC(`$o2$`)--`, ...`RC(`o_n`)++`, but only `RC(`o_1`)--` and `RC(`o_n`)++` are needed. They exploit this observation in their *Sliding Views* collector. Memory management operations are divided into cycles. A mutator writing a pointer first checks the source object's dirty bit: if it is clear, it is set and the address of slot and its old value are added to a local buffer. At the end of each cycle, each mutator thread marks objects referenced from its stack and local buffer, clears the dirty bits and passes its local buffer to the collector before resuming. The collector processes the mutator buffers, using the old values of the pointers to perform the reference count decrement and the values currently held in the object to perform the increment. However, he algorithm is complex: it uses four handshakes to synchronise correctly with concurrent mutator activity. Paz et al.'s *Age-oriented* collector [51] collects all generations in heap at every collection but, in the spirit of [12] traces the young generation, and uses sliding views to collect the old generation.

Bacon & Rajan [5] demonstrate a high-performance, concurrent, cycle-reclaiming, reference counter. As with any efficient reference counting implementation, it uses deferred RC. Reference counting operations are managed by a producer-consumer model: mutators add RC operations to local buffers. These buffers are periodically turned over to a collector running on its own processor which consumes them: only the collector thread can modify reference counts. Garbage collection is divided into epochs: increment operations are done in this epoch but decrements are deferred to the next to avoid race conditions. Cycles are handled by local rather than global tracing using trial deletion [18]. Starting from all non-atomic objects whose reference count has been deleted, the graph is traced and a second reference count of each object encountered is decremented, in order to remove the effect of pointers internal to a cyclic garbage structure. If, at the end of this phase, all of the objects encountered have a zero secondary reference count, the entire structure is reclaimed. Bacon & Rajan found that throughput was close to that of a parallel mark-sweep collector for most applications, with maximum pause times of 2.6ms and the smallest interval between pauses of 36ms. One problem for concurrent trial deletion is that a thread may alter the connectivity of the graph being traced. Paz et al. [50] resolve this by applying sliding views to Bacon & Rajan's collector.

6 Real-time collection

The collectors we have examined so far have sought to meet only soft real-time bounds, with most pause times less than a threshold. Collectors for hard real-time systems must meet more stringent requirements: all deadlines must be met. However, note that simple bounds on the maximum length of any garbage collection pause are insufficient. Were a collector to incur a large number of such pauses within a short period, it is possible that the mutator's share of the processor would be insufficient. For this reason, *minimum mutator utilisation* curves[5] [16] provide a superior metric to maximum pause time or, better, pause time distributions. MMU curves plot the mutator's minimum utilisation of the processor in any time slice (see an example in Figure 2). Other issues for hard real-time collection include handling of large data structures, fragmentation and schedulability.

One solution is to provide the programmer with regions that allow fast allocation but that can also be reclaimed in constant time (i.e. all objects within a region are deallocated at once); one example is the *Real-time Specification for Java*. However, this is a cumbersome model to program with as stack-based region discipline imposes restrictions on the direction of pointers between regions: often data shared between regions must be passed via the general heap, thus undermining the region strategy. Standard libraries also have to be rewritten to be region-aware. In contrast, the *Metronome* [6] provides a general purpose collector that can meet hard real-time bounds *provided* that the programmer is able to specify parameters such as the maximum allocation rate of the program.

The Metronome allocates objects from segregated free-lists [68], using a large number of size classes

[5]See a discussion of garbage collection metrics in [56].

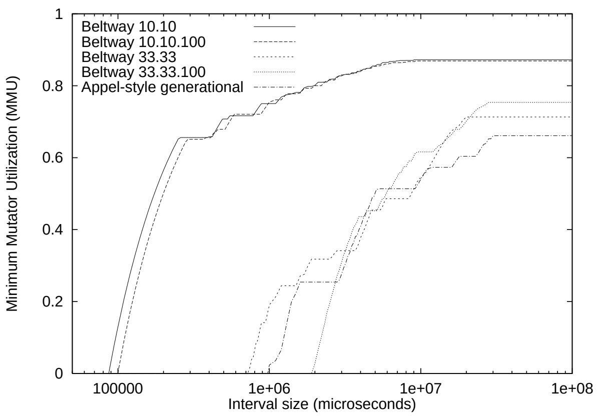

Figure 2: MMU curves (for Beltway configurations running SPECjvm98 _213_javac). The *x*-axis intercept gives the maximum pause time, the *y* intercept the total fraction spent in garbage collection; higher and lefter-most is better.

to achieve low internal fragmentation; external fragmentation is removed by copying (although the collector is mostly non-copying). Large arrays are broken into 2-level structures (a sequence of power-of-2 sized arraylets), with a limit on maximum array size (e.g. 8MB). Compiler optimisations improve access to these structures. Metronome uses an incremental marking, lazy sweeping collector with a snapshot at the beginning write barrier to avoid retracing. Care is taken to limit fragmentation. If too few free pages are available, objects are copied from a fragmented page to another of the same size class. A black-only read barrier is used to ensure that the mutator sees only toSpace objects. The collector is scheduled on the basis of time-based quanta as work-based quanta are found not to give consistent mutator utilisation. On SPECjvm98 benchmarks, for a 500MHz PowerPC, with average allocation rate 10–19MB/s and 80–358MB/s maximum, maximum live data 20–34MB, Metronome provided a minimum mutator utilisation of 44% and pause times in the range 10–13ms; less than 2% of data was copied rather than traced.

7 Heap organisation

So far we have discussed methods of improving the speed with which memory management operations can be implemented. However, another way to improve performance is to reduce the amount of work that the collector has to do by exploiting knowledge of the lifetime demographics of objects.

Generational garbage collection, based on the hypothesis that ‘most objects die young’ [63], segregates objects by age into generations. Generational collectors concentrate effort on the region (the young generation) where least live data is expected and avoid repeatedly processing long-lived objects in the old generation. An alternative strategy is to give objects as much time as possible to die. *Older-first* garbage collection [60] lays out objects in the heap in allocation order and then collects them in age order (see Figure 3). Starting from the oldest objects in the heap (say, the left of the heap), at each collection an older-first collector chooses a window to be collected immediately to the right of the of the survivors of the last collection. With each collection, this window sweeps rightwards. Eventually it reaches the youngest (or right hand) end of the heap and is reset to the oldest (left hand) end of the heap. While older-first collection is an interesting proposition, its performance is modest.

The *Beltway* garbage collection framework [11] is novel in that it separates incrementality and age. It generalises all known copying collectors as well as providing new ones. The unit of collection is the *increment*. Increments are held on *belts*; a belt may have one

Figure 3: Older-first garbage collections. At each collection, the next younger region (grey) is collected; survivors are shown black.

Figure 4: A beltway collector configured with 3 belts. Belt 2 has one increment, belts 0 and 1 each have two increments.

or more increments, but these are collected in strictly FIFO order, lowest-numbered belt first (see Figure 4). Survivors may be copied to another increment on the same belt or promoted to a higher-numbered belt (c.f. generations). Increments provide incrementality: allowing a belt to have more than one increment gives objects time to die. The *LACE* framework [36] extends Beltway to exploit lifetime patterns even further. It associates belts with collection policies (e.g. expected object lifetimes). Thus, memory allocated by a site expected to create long-lived objects would be on a belt each of whose increments are set not to be processed for a long time after its creation. Belt policies also allow different promotion patterns, for example a promotion path might be from a short-lived belt to an immortal belt, reflecting a surprisingly common demographic pattern; data allocated onto this pathway would be processed by the collector at most once.

8 Integration into the environment

Good memory manager performance depends on good integration with the manager's environment. There are two aspects to this. First, collectors should take advantage of expected program behaviour, as discussed above. Secondly, the collector should inter-operate well with the operating system and hardware. Typically, this means at least avoiding adverse effects on locality and at best reorganising data to improve locality [66]. Until recently, the disparity in processor and memory speeds have grown ever steeper year by year as improvements in processor speeds were not matched by improvements in memory speeds. Tracing garbage collection disrupts a program's working set as it touches all live data. Additional paging kills performance, and contention between mutator and collector for the cache is undesirable and likely to become a more important factor as the number of cores on a chip increases.

Over-provisioning memory is unlikely to be a realistic solution for many users. Consumer PCs are commonly sold with barely adequate memory, yet over the lifetime of the machine the memory demands of the programs run on it increase. Solutions to avoid paging such as mark-bitmaps [13] or hierarchical copying [44] have been known for a long time. However, from the demise of Lisp machines until recently, no systems have explored better cooperation between collector and operating system.

The *Bookmarking* collector [33] attempts to eliminate garbage collection-induced paging. In particular, the collector does not touch evicted pages whilst still providing an approximation to full heap collection. The collector is generational, with a mark-sweep old generation and occasional compaction (into empty slots in segregated free-lists). The heart of the system lies in cooperation between the collector and a modified Linux virtual memory manager over which pages to evict. The collector reacts to signals from the virtual memory manager that pages are scheduled for eviction or made resident. On notice of eviction, the collector attempts to select an empty but resident page (e.g. a fromSpace page immediately after a collection, although recently used, would no longer be required). Otherwise, the collector scans the victim page and remembers its outgoing pointers (bookmarks) before protecting the page and notifying the virtual memory manager that it can be evicted. On the page's return to main memory, mutator access violations are handled by removing the bookmarks for this page before unprotecting it. Garbage collections proceed as normal but references to evicted pages are not followed; bookmarked objects are treated as root-referenced. Other approaches monitor paging behaviour in order to guide the heap manager's decisions whether to expand the heap or trigger a collection [71, 69], for example by instrumenting the code at phase boundaries.

9 Static analysis

Whether through our instincts as programmers or through exhaustive examination of program traces, it is clear that programs exhibit particular patterns of memory behaviour and that many of these patterns are robust against changes of input. It is therefore very appealing to ask whether these patterns can be identified and exploited automatically.

Region inference has been shown to be an effective way of managing memory in the ML-kit [62]. However, without a deep understanding of the inference algorithm, it is difficult to track down leaks and other memory problems. Programs may need a back-up tracing collector [31]. Region inference also does not allow individual objects to be deallocated. In contrast, the combination of pointer analysis and liveness analysis of Guyer et al. [30] identifies points in a program when individual objects (rather than regions or all objects created by a single allocation site) may be freed. Although this give better throughput than using a mark-sweep collector, it was unable to improve over a generational mark-sweep collector. It was also ineffective for long-lived objects, container internals and classes that behaved *mostly* like factories. Cherem & Rugina [17] track reference counts statically to insert free operations, guarded by suitable predicates. Their technique was able to free a large proportion of data allocated for some programs, and showed some execution time improvements over mark-sweep on small heaps.

Static analyses have also been used to assist the collector rather than directly reclaim space. Guyer & McKinley combine a flow insensitive pointer analysis with a runtime test to pretenure a new object in the old generation if it will be referenced by an object already in the old generation. Hirzel et al. [34] use a conservative compiler analysis to derive connectivity information in order to partition objects in the heap. The intention is to replace the directed graph of objects with a tree of strongly connected components. The collector can then reclaim subsets of the partitions without need for a write barrier.

10 Conclusions

The pace of development of new architectures, new environments and new applications combined with the increasing importance of languages supported by managed runtimes means that memory management research is as alive and as vital as ever. The challenges for the future include the development of high performance collectors for the multicore, multiprocessors that will be prevalent. Collectors will need to interact better with their environment, taking account of the virtual memory manager, cache and energy considerations. The efforts of memory managers will become better directed, taking more account of program behaviour, with richer collectors managing different spaces under different policies, possibly guided by program analyses.

References

[1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. In *OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[2] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In OOPSLA [46], pages 314–324.

[3] Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen Fink, David Grove, and Ton Ngo. Experiences porting the Jikes RVM to Linux/IA32. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '02)*, San Francisco, CA, August 2002.

[4] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[5] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In PLDI [53].

[6] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[7] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[8] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[9] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *ICSE 2004, 26th International Conference on Software Engineering*, Edinburgh, May 2004.

[10] Stephen M. Blackburn and Tony Hosking. Barriers: Friend or foe? In Diwan [25].

[11] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In PLDI [54], pages 153–164.

[12] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In OOPSLA [47].

[13] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[14] *Proceedings of the 14th International Conference on Compiler Construction*, Edinburgh, April 2005. Springer-Verlag.

[15] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[16] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In PLDI [53], pages 125–136.

[17] Sigmund Cherem and Radu Rugina. Compile-time deallocation of individual objects. In Moss [45], pages 138–149.

[18] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.

[19] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In Vitek [65].

[20] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[21] David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. Garbage-first garbage collection. In Diwan [25].

[22] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[23] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[24] Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.

[25] Amer Diwan, editor. *ISMM'04 Proceedings of the Fourth International Symposium on Memory Management*, Vancouver, October 2004. ACM Press.

[26] Daniel J. Edwards. Lisp II garbage collector. AI Memo 19, MIT AI Laboratory, Date unknown.

[27] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[28] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.

[29] Alex Garthwaite, Dave Dice, and Derek White. Supporting per-processor local-allocation buffers using lightweight user-level preemption notification. In Vitek [65].

[30] Samuel Z. Guyer, Kathryn McKinley, and Daniel Frampton. Free-Me: A static analysis for automatic individual object reclamation. In PLDI [55], pages 36r–375.

[31] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In PLDI [54], pages 141–152.

[32] Timothy P. Hart and Thomas G. Evans. Notes on implementing LISP for the M–460 computer. In E. C. Berkeley and Daniel G. Bobrow, editors, *The Programming Language LISP: Its Operation and Applications*, pages 191–203, Cambridge, MA, 1974. Information International, Inc.

[33] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of SIGPLAN 2005 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Chicago, IL, June 2005. ACM Press.

[34] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In OOPSLA [47].

[35] Xianlong Huang, J. Eliot B. Moss, Kathryn S. McKinley, Stephen M. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java virtual machines. Technical Report TR–03–03, University of Texas at Austin, February 2003.

[36] Richard Jones and Chris Ryder. Garbage collection should be lifetime aware. In Olivier Zendra, editor, *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)*, page 8, Nantes, France, July 2006.

[37] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[38] Haim Kermany and Erez Petrank. The Compressor: Concurrent, incremental and parallel compaction. In PLDI [55], pages 354–363.

[39] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, volume 22(7) of *ACM SIGPLAN Notices*, pages 253–263. ACM Press, 1987.

[40] Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[41] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of *ACM SIGPLAN Notices*, Tampa, FL, October 2001. ACM Press.

[42] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[43] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[44] David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, August 1984. ACM Press.

[45] J. Eliot B. Moss, editor. *ISMM'06 Proceedings of the Fourth International Symposium on Memory Management*, Ottawa, June 2006. ACM Press.

[46] *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, Denver, CO, October 1999. ACM Press.

[47] *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[48] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In PLDI [54], pages 129–140.

[49] Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. Mostly concurrent compaction for mark-sweep GC. In Diwan [25].

[50] Harel Paz, Erez Petrank, David F. Bacon, V.T. Rajan, and Elliot K. Kolodner. An efficient on-the-fly cycle collection. In CC [14].

[51] Harel Paz, Erez Petrank, and Stephen M. Blackburn. Age-oriented garbage collection. In CC [14].

[52] Pekka P. Pirinen. Barrier techniques for incremental tracing. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 20–25, Vancouver, October 1998. ACM Press.

[53] *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[54] *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[55] *Proceedings of SIGPLAN 2006 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Ottawa, June 2006. ACM Press.

[56] Tony Printezis. On measuring garbage collection responsiveness. *Science of Computer Programming*, 62:164–183, October 2006.

[57] Manuel Serrano and Hans-J Boehm. Understanding memory allocation of Scheme programs. In *Proceedings of International Conference on Functional Programming*, Montreal, September 2000. ACM Press.

[58] Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

[59] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[60] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In OOPSLA [46], pages 370–381.

[61] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, December 1991.

[62] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3), September 2004.

[63] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[64] Martin Vechev, Erin Yahav, and David Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In PLDI [55], pages 341–353.

[65] Jan Vitek, editor. *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, Chicago, IL, June 2005. ACM Press.

[66] Jon L. White. Address/memory management for a gigantic Lisp environment, or, GC Considered Harmful. In *Conference Record of the 1980 Lisp Conference*, pages 119–127, Redwood Estates, CA, August 1980.

[67] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[68] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[69] Ting Yang, Emery D. Berger, Matthew Hertz, Scott F. Kaplan, and J. Eliot B. Moss. Autonomic heap sizing: Taking real memory into account. In Diwan [25].

[70] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

[71] Chengliang Zhang, Kirk Kelsey, Xipeng Shen, Chen Ding, Matthew Hertz, and Mitsunori Ogihara. Program-level adaptive memory management. In Moss [45], pages 174–183.

[72] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, November 1990.

[73] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, March 1989. Technical Report UCB/CSD 89/544.