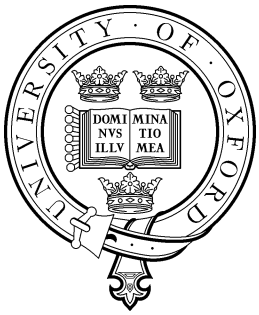
Six Myths and Paradoxes of Garbage Collection



Holly Cummins
Kellogg College
University of Oxford

A dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering

April 17, 2007

Acknowledgements

David Siegwart and Andy Wharmby, my local garbage collection experts, provided valuable discussion and many helpful suggestions. I am indebted to Dicken Winyard for taking the time to photograph garbage using his serious studio lighting; all photographs are taken by him. Plots were produced using a modified version of the Extensible Verbose Toolkit (available as part of the IBM Support Assistant).

My supervisor, Jim Davies, provided helpful feedback and coaxed me into getting on with the business in hand and producing early drafts.

The author confirms that: this dissertation does not contain material previously submitted for another degree or academic award; and the work presented here is the author's own, except where otherwise stated.

Abstract

Many myths and paradoxes surround garbage collection. The first myth is that garbage collection is only suitable for the incompetent, unskilled, or lazy. In fact garbage collection offers many architectural and software engineering advantages, even to the skilled developer. The second myth is that garbage collection is all about about collecting garbage. Garbage collectors also include an allocation component, which, along with their powers of object rearrangement, can make a significant difference to application performance. Thirdly, criticisms of garbage collection often focus on the pause times, and responses to these criticisms often focus exclusively on reducing pause times, in the mistaken belief that small pause times guarantee good application response times. Pause times are also often used as a metric of general application performance, and an increase in pause times is taken as an indicator of worsened performance, when in fact the opposite the opposite is often true. Paradoxically, even the total amount of time spent paused for garbage collection is not a good predictor of the impact of garbage collection on application performance. Finally, the sixth myth is that garbage collection has a disastrous performance impact. While garbage collection can hurt application performance, it can also help application performance to the point where it exceeds the performance with manual memory management.

Contents

List of Figures

Chapter 1

Introduction

Here's a good thing to do if you go to a party and you don't know anybody: First take out the garbage. Then go around and collect any extra garbage that people might have, like a crumpled napkin, and take that out too. Pretty soon people will want to meet the busy garbage guy.

Jack Handy, Deep Thoughts, *Saturday Night Live*

Garbage collection is a system of automatic memory management. Memory which has been dynamically allocated but which is no longer in use is reclaimed for future re-use without intervention by the application. Garbage collection solves the otherwise difficult problem of determining object liveness by freeing memory only when it becomes unreachable.

Garbage collection is now very widely used in modern languages. Garbage collected languages include Java, the .Net languages, Lisp, Python, Perl, PHP, Ruby, Smalltalk, ML, Self, Modula-3, and Eiffel. Some languages which are not traditionally garbage collected offer garbage collection as a pluggable or configurable extension. For example, collectors are available for C++, and Objective-C was recently extended to allow garbage collection.

Garbage collection has been the subject of much academic research [50, 89]. In particular, the volume of new techniques with various claimed properties is high enough that it is tempting to coin the phrase YAGA (Yet Another GC Algorithm) [18, 51, 78, 58, 7, 32, 25].

1.1 Garbage collection algorithms

Despite the large number of variants, most garbage collection algorithms fall into a few simple categories.

1.1.1 Reference counting

Reference counting garbage collectors track the number of references to each object. Often the count is maintained in and by the object itself. When a new reference to an object is added, the reference count is incremented. When a reference is removed, the count is decremented. When the count reaches zero, the object is destroyed and the memory released. Reference counting fails when objects reference one another in a cycle. In these cases each object will be seen to be referenced and will never be freed, even if nothing references the cyclic structure. Reference counting is unable to deal with cycles unless augmented with occasional invocations of another form of garbage collection. Reference counting has a number of other disadvantages, mostly to do with performance and bounding of work, and so it is rarely used in modern garbage collection systems. However, smart pointers in C++, while not traditionally considered a form of garbage collection, do make use of reference counting. Ad hoc

garbage collectors introduced to manage object reachability in complex environments also tend to use reference counting since it is easily implemented in a hostile environment.

1.1.2 Tracing collectors

Most garbage collectors perform some sort of tracing to determine object liveness. Tracing does not have the same problem with collecting cycles as reference counting does. However, tracing collectors are more sensitive to changes in the object graph and usually require the suspension of the application threads or close monitoring of application activity to ensure correctness.

Mark-sweep

Mark-sweep collectors collect in two phases. In the first phase, all reachable objects are traced, begining from a set of roots. The roots are all objects which are guaranteed reachable, including objects referenced from the stack and static variables. Every object directly or indirectly reachable from a root is marked. Objects which do not end up marked are unreachable and therefore cannot be used again by the application, so they may safely be freed.

In the second phase of a mark-sweep collection, all unmarked objects are added to a free-list. Contiguous areas of free memory are merged on the free list.

Mark-compact collectors

Mark-compact mark live objects using identical techniques to those used by mark-sweep collectors. However, mark-compact collectors do not use a free list. Instead, in the second phase all reachable objects are moved so that they are stored as compactly as possible. New allocation is then performed from the empty area of the heap.

Many collectors hybridise these two approaches, combining frequent sweeping with occasional compaction.

Copying collectors

Copying collectors divide the heap into two areas, a to-space and from-space. All objects are allocated in the to-space. When the new space is full, a collection is performed and the spaces are swapped. All reachable objects in the to-space are copied to the from-space, which is declared the new to-space. New allocation is then performed in the new to-space.

Copying collectors have a number of advantages over marking collectors. Because copying is done in the same phase as tracing, it is not necessary to maintain a space-consuming list of which objects have been marked. Because no sweep is performed, the cost of the collection is proportional only to the amount of live data. If most objects are unreachable, a copying collecton can be very efficient. The corollary is that if most objects remain reachable, or if some very large objects remain reachable, the collection will be very inefficient, because a large amount of memory will need to be copied.

Copying collectors also keep the heap very compact, and this can boost allocation and application performance. However, because a completely empty from-space must be maintained at all times, copying collectors are not space-efficient. Modern collectors tend to estimate an object death-rate and maintain less than half the heap for the from-space accordingly. Copying collectors are also known as semispace collectors.

1.2 Garbage collection flavours

There are many variations on these basic classes of collector.

1.2.1 Generational collectors

Generational collectors exploit the observed properties that most objects tend to die young, and that young objects are more likely to reference old objects than the reverse. Together these are known as the weak generational hypothesis.

Generational collectors divide the heap up into multiple generations. Young generations can be collected without collecting old generations. These partial collections are quicker than full heap collections, and are likely to produce a good return of free space relative to the area collected, since most objects die young. At least the younger generations tend to employ copying collectors, since these collectors are very efficient in heaps with high attrition rates.

Objects which survive a given number of collections are tenured and move up to an older generation. In order to avoid collecting objects in younger generations which are referenced by older generations, a remembered set is maintained of references from old generations to young generations. A write barrier between the generations is used to catch changes to these references.

1.2.2 Incremental collection

Incremental collectors allow single collections to be divided into smaller collections. This allows pause times to be limited. Incremental techniques almost always require one of a write barrier or a read barrier to prevent changes to the object connectivity graph mid-way through a collection causing the reachability results to be incorrect.

1.2.3 Concurrent collection

A concurrent collector is one which can execute concurrently with a mutator thread. Even on multi-processor systems, it usually does not make sense to dedicate an entire processor to running garbage collection. Therefore the concurrent collection is perhaps more accurately described as a highly incremental collection; each thread is assigned small units of garbage collection work to do along with its application work, so the garbage collection work is finely interleaved with application work. As with incremental techniques, concurrent collections need a write- or read-barrier.

1.2.4 Parallel collection

A parallel collector is one which divides the collection work so that multiple collector threads are collecting concurrently. This is counter-productive on single-threaded processors, but increasingly necessary as the number of processors in a system increases. On systems with many processors a single-threaded collector is unable to keep up with the amount of garbage the processors can produce.

Similar techniques are used to achieve incrementality, concurrency, and parallelism, so many collectors which have one of these properties also have some of the others.

1.3 Other concepts

Any discussion of garbage collection requires some other concepts unrelated to particular algorithms.

1.3.1 Roots

Garbage collection roots are objects which are known or assumed to be live. Direct and indirect references from the roots to objects in the heap are used to determine object liveness in tracing collectors. (Reference counting collectors are not tracing collectors, but all other collectors are.) Figure 1.1 illustrates a set of roots and the objects which are and are not counted as live on the basis of those roots.

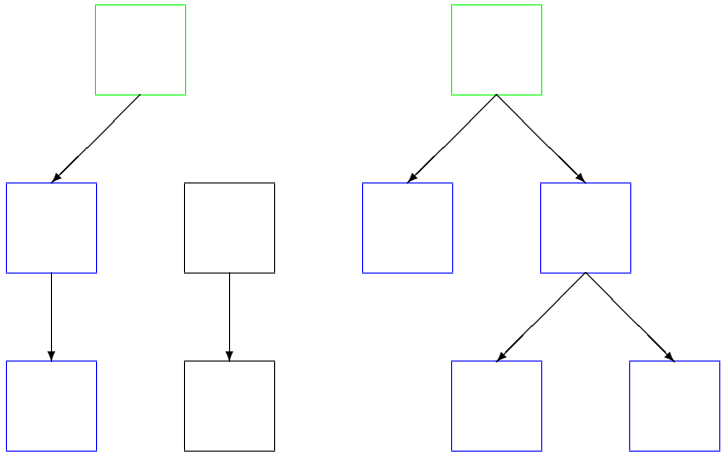


Figure 1.1: A set of garbage collection roots and objects. The roots are coloured green. Objects which are coloured blue are directly or indirectly references from the roots and are considered live. The black objects are unreachable from the roots and are candidates for garbage collection.

1.3.2 Mutators

In the garbage collection literature, the application threads are usually referred to as mutator threads. From the perspective of the garbage collector, the function of the application threads is to inconsiderately rearrange the contents of the heap, while the garbage collector does the "real" work.

1.3.3 Stop the world

Garbage collection algorithms fall into two categories, concurrent and non-concurrent. In non-concurrent algorithms all mutator threads must be stopped. These pauses are known as stop-the-world pauses.

1.3.4 Verbose GC

Many languages with garbage collection facilities provide a verbose mode for the garbage collector. The logs produced in the verbose mode record details of heap usage and collection pause times. While the logs are undoubtedly helpful, superficial analysis of the pause time logs is sometimes highly misleading. This is the subject of chapters 5 and 6.

1.3.5 Objects

The term "object" is used throughout to refer to program structures. These are not necessarily objects in an object-oriented sense, since garbage collection does not imply object orientation.

1.4 A note on benchmarks

A small number of languages and benchmarks are used to illustrate the examples here. In particular, the SPECjbb benchmarks are used heavily. SPECjbb is a benchmark produced by the Standard Performance Evaluation Corporation, and it was developed in cooperation with a range of industry and academic representatives [27]. While the SPEC benchmarks have been criticised [16], they are an accepted industry standard for Java performance benchmark. Like all benchmarks, they are intended to simulate common scenarios and produce performance results which are representative of a typical application. Like all benchmarks, the true representativeness of their results is also subject to debate. The issue is not so much any particular characteristic of the SPEC benchmarks as the breadth of characteristics of real applications. Even different versions of the same benchmark, such

as the SPECjbb 2000 and SPECjbb 2005 benchmarks, can have wildly different performance characteristics on different systems. Some applications are very sensitive to small system and language differences, and this sensitivity is unlikely to be echoed in popular benchmarks. The corollary of this is that some benchmarks have idiosyncratic sensitivities or insensitivities, and these are unlikely to be echoed in many other applications. For example, SPECjbb 2000 had a marked insensitivity to heap fragmentation and compaction times which was echoed in almost no real application [27].

For these reasons, benchmark results must be interpreted with caution. They illustrate possible application performance results, rather than universal ones. The only entirely reliable way to predict the effect of parameter, system, or vendor changes for a given application is to test those changes for the application.

Benchmark results are also subject to statistical variation. Even runs of the same benchmark with the same parameters have limited reproducibility. For some scenarios, system architectures, and garbage collection policies, the amount of variation can be surprisingly high. While benchmark results are generally published for a single run with no statistical analysis, a proper exploration of the performance of a system or algorithm should include multiple runs. The performance results presented here are intended to be illustrations and counter-examples, rather than proofs, and so statistical analysis is generally omitted.

Chapter 2

Garbage collection is only for the incompetent, unskilled, or lazy

A mental disease has swept the planet: banalization. Everyone is hypnotized by production and conveniences — sewage systems, lifts, bathrooms, washing machines. [. . .] Presented with the choice of love or an automatic garbage disposal unit, young people of all countries have chosen the garburetor.

Ivan Chtcheglov, *Formulaire Pour Un Urbanisme Nouveau*

Garbage collection automates program memory management. This reduces the amount of time required to develop an application, and also eliminates a whole class of common programmer errors. The corollary of this is that development requires less effort, less diligence, and a lower skill level. Because of this garbage collection is sometimes characterised as being only required by, or even as only suitable for, the inexpert, inept, or indolent.

It is difficult to provide evidence for this assertion based solely on a survey of the garbage collection literature; garbage collection has been the subject of almost universally positive academic interest. Outside academic contexts, however, garbage collection is often treated in a much less positive light.

Why is the academic discussion so much more positive than the non-academic one? While it is tempting to believe this is because academics are much too clever to fall foul of any of the myths discussed here, it is more likely simply that most garbage collection research is conducted by people favourable to the idea of garbage collection. The academic treatments are generally by people working in the field of garbage collection who have some positive emotional investment in the subject. The general development community, on the other hand, have no such interest, and many in fact have an interest in avoiding lowering the bar for entry to their field.

2.1 Manual memory management: lye soap for the soul

A survey of the less formal corners of the internet gives rich anecdotal evidence that dismissive attitudes to garbage collection persist. It would seem that eschewing garbage collected languages is good for the soul, much like using abrasive soap or keeping the central heating turned off.

In a response to a blogger who is considering starting his own IT company [68] , a reader advises ‘avoid using any language that uses “garbage collection” memory management. It teaches bad habits and makes for lazy, sloppy engineers.’ Similarly, in a discussion of the inclusion of GC in the D language [4], a poster to the SkyOS message board writes ‘when programmers rely on a GC implementation to clean up their mess, they become lazy. ’

A particularly recent illustration of contemporary attitudes to garbage collection can be found in an August 2006 slashdot thread responding to an announcement that the Objective-C language would support garbage collection [47]. The Objective-C language is the preferred language for systems programming on Mac OS X. While many responded positively, one distraught poster titled his comment ”NOOOOOOO #@$#$@”. He went on to write

> Garbage collection is a step backward, IMO, but every language seems to be moving in this direction. I really do believe that resource awareness is crucial to efficient programming. Garbage collection encourages lazy programming habits, which I've seen in quite a few Java developers.

In another comment titled "Hacks and Novices Rejoice!", a user mocks the literacy and understanding of garbage collection users:

> "Garbage Collection is cool cuz you don't have to, like, remember to delete stuff"
> Shudder.

Even some of the more moderate comments suggest that garbage collection is irrelevant to highly skilled developers. One writes 'At the end of the day Apple aren't really catering for the top programmers here who will write good programs no matter what. It'll help the rest write better programs though.'

The idea that garbage collection is intended for use by those with modest intellectual capacity is echoed by essayist Paul Graham in his comments on Java[42]:

> Like the creators of sitcoms or junk food or package tours, Java's designers were consciously designing a product for people not as smart as them. [...] It's designed for large organizations. Large organizations have different aims from hackers. They want languages that are (believed to be) suitable for use by large teams of mediocre programmers– languages with features that, like the speed limiters in U-Haul trucks, prevent fools from doing too much damage.

Similar sentiments crop up in regular debates on the comp.lang.c++.moderated usenet newsgroup, with statements such as "People who prefer GC tend to be difficult to grasp algorithms [sic]" [37]. GC users are categorised into a group whose other deficiencies include a lack of rigour, no capacity for design, and even a fundamental lack of big imagination [67]. GC users are sometimes even grouped with the developmentally subnormal:

> I myself am not retarded, so I've no need for "Garbage Collection". If, hypothetically speaking, I forsaw that I would temporarily become retarded (a golfclub to the head maybe), then I would make use of auto_ptr, but that has yet to happen. [66]

Some go further and argue that garbage collection is unnecessary even as a crutch for the dimmer members of the development community. In the comp.lang.c++ newsgroup, Claudio Puviani [75] writes

> If you want garbage collection, you can implement it yourself or use someone else's garbage collector. But why would you want to? There are plenty of safe and fast memory management techniques in C++ that complete [sic] obviate the need for automatic and implicit memory management.

Not only is not needed, garbage collection is an fundamental abdication of a developer's responsibility. In a blog entry discussing C#, Juha Niemenen [65] describes using garbage collection as 'the coder leaves it to the garbage collection engine to clean up his memory allocation mess.' On the problem of ambiguous deallocation responsibilities when objects are passed across module boundaries, Nieminen confidently asserts "If the problem is in crappy OO design, the right solution is not to encourage doing it by fixing the problem with a compiler feature."

He continues, in response to the argument that garbage collection improves program correctness,

> A program does not become magically more "correct" if the compiler and/or interpreter fixes the programmer's mess (which is caused by a poor OO design).

2.1.1 Effort reduction

Does popular opinion on the internet have credibility — is using automatic memory management really a symptom of laziness? If laziness is defined as doing less work, the answer does in fact appear to be 'yes'.

It is certainly widely acknowledged by both advocates and opponents of garbage collection that it can significantly reduce the amount of developer time spent tracking down and correcting memory management errors. Somewhat surprisingly, there has been little peer-reviewed research attempting to quantify this productivity gain. The clearest results were published by Rovner. Rovner compares two Xeroc PARC-developed Pascal derivatives, Mesa and Cedar [77]. Cedar is based on Mesa but has automatic memory management. He claims that developer time spent on storage management problems decreased from about 40% of total time in Mesa to about 5% in Cedar. Rovner's data is particularly compelling since the compared languages were identical except in their memory management strategies.

Later comparisons have generally focussed on comparisons between Java and C++. Phipps used the Personal Software Process [54] to experimentally measure his productivity on two projects, one C++, and one Java [70]. Despite being an experienced C++ programmer and a relatively inexperienced Java programmer, he found that the C++ programs had two to three times the defects per line of code as the Java programs. He also found they took twice as long to debug and correct. He speculates the main cause for the differences in defect rates was the memory management model in Java, but his defect tracking was not detailed enough to provide exact statistics on the fraction of C++ defects which were caused by memory management mistakes. The scope of Phipps' experiment was limited to one programmer and two different projects. However, each project lasted several months, so the number of lines of code included was significant.

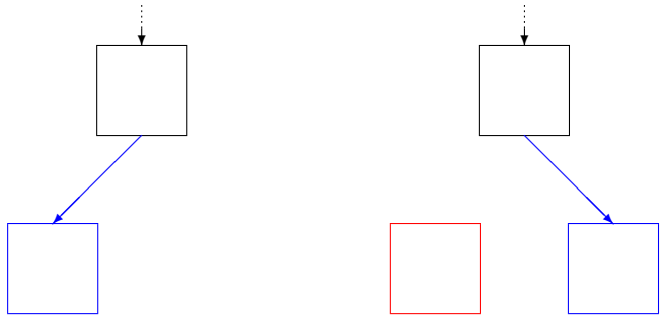
A much more broad-ranging study, involving eighty implementations of the same specification and seventy four programmers, was conducted by Prechelt. The limitation of Prechelt's experiment was that the programming exercise was much simpler than the ones considered by Rovner or Phipps. Both development-time and execution time were short enough that a large class of memory management problems would not be encountered. Prechelt studied the relative productivity of developers solving the same problem in seven different languages, C, C++, Java, Python, Perl, Rexx, and Tcl [74]. Of these languages, only C and C++ require manual memory management, while the others all use some form of automatic memory management. He observed a general correlation between development time and manual memory management, but it was less pronounced than variations in developer ability within each language. For example, three of the Java developers ran into difficulties and had work times of 40, 49, and 63 hours. These are dramatically longer than the median development time among the remaining Java developers of around 10 hours.

Over all languages, the median work time for the scripts was 3.1 hours, and the median work time for the non-scripts was 10.0 hours. The scripts had a clear advantage over the non-script languages in terms of developer productivity, but it is not clear how much of this advantage can be attributed to the automatic memory management. Comparison of Java, C, and C++ suggests that the other characteristics of the scripted languages may be responsible for the difference. While the development times for Java were one or two hours less than those for C++, the C development times were in fact a further one or two hours faster than the Java development times – even when the Java outliers were removed.

The development time which is saved by using automatic memory management need not necessarily be squandered on the internet or down at the pub by slothful developers. Boehm points out that the saved development effort could be used for performance tuning [19]. He suggests that using manual memory management is one form of trading developer time for performance, and other uses of that time may give much more substantial performance gains.

2.1.2 Error reduction

It may be, however, the the main advantage of garbage collection is not the time taken to produce a program, but the time taken to produce a *correct* program. By this metric, Prechelt's results seem to confirm the advantage of memory management but, as before, it is difficult to rule out the influence



(a) A new object is allocated.

(b) A second object is allocated and the reference to the original object is overwritten. The original object can be neither used nor freed.

Figure 2.1: One common way in which memory leaks are created.

of other language factors. All of the programs passed an acceptance test before being submitted, but many of them failed when faced with variation of the input data. Prechelt categorised some as highly faulty, since they timed out or failed to load the larger dictionary, or produced incorrectly formatted output more than half the time. 13% of the C/C++ programs fell into this category, along with 8% of the Java programs, and 5% of the scripts. The languages with memory management were more likely to be correct, but the difference was relatively minor. On the other hand, the programs tested were very short-lived (a few seconds to a minute), and so memory leaks would be unlikely to have any effect. Were the programs to be run continuously for several days, the failure rate of the C/C++ programs might increase significantly. Similarly, more complex program invocations or longer execution paths might expose more dangling pointers.

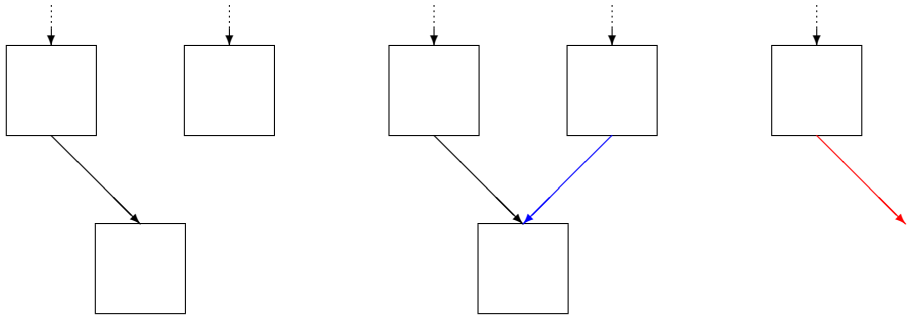
In general garbage collection reduces or eliminates two classes of errors, memory leaks and dangling pointers. Dangling pointers are more severe, but memory leaks can be serious and also extremely difficult to detect.

Memory leaks

If allocated memory is not correctly deallocated, a program will be subject to memory leaks. If pointers to allocated memory are overwritten or go out of scope, it becomes impossible to reclaim the memory until the program terminates. In long running programs, the unreclaimed memory can accumulate until no more memory is available for the program's use. Some embedded systems do not automatically free application memory even when an application exits, and so memory leaks are serious even in short-running programs on these systems. Leaked memory will not be available for re-use until after a reboot. Figure 2.1.2 illustrates leaked memory.

Memory leak detection is the subject of active academic research [46, 45, 97]. Leaks can be detected by static analysis, but with some risk of false positives and missed leaks. Leaks can also be detected by runtime analysis which replaces conventional allocation and deallocation functions with ones which track invocations and produce a report on unmatched allocations. Some tools use a garbage collector to trace references and highlight any collectable objects as memory links [44]. Many tools exist which exploit combination of these techniques; popular tools for C and C++ include Rational's Purify [44], mtrace, memwatch, dmalloc [39], ccmalloc, NJAMD, valgrind, mpatrol, Parasoft Insure++ [38], mprof [99], LeakTracer [1], SWAT [45], Etnus TotalView, and deleaker. The list of tools is long enough that there is even one called YAMD (yet another malloc debugger) [36].

Memory leak detection tools are so numerous because memory leaks are ubiquitous. Xie and Aiken tested a static analysis tool on six large open source projects and detected 455 unique memory leaks [97]. They found 82 leaks in the linux kernel, 117 in OpenSSL, and 136 in the binutils package. Measured relative to project size, OpenSSH had the greatest number of leaks, at nearly 0.8 leaks per



(a) An object holds a reference to another object

(b) A second object is given a reference to the same object.

(c) One of the parent objects is freed and frees its children in order to avoid memory leaks. The other reference to the child is left dangling.

Figure 2.2: Creation of a dangling pointer.

thousand lines of code, with OpenSSL having 0.4 leaks per thousand lines of code. Despite having 82 leaks in total, the linux kernel was actually quite well behaved relative to its size, with only 0.016 leaks per thousand lines of code. The next most robust was Postfix, with 0.06 leaks per thousand lines of code. Even the lower memory leak counts are surprisingly high, however.

Xie and Aiken classify the leaks into three types; missed deallocation on error paths, covert allocators, and misunderstood function interfaces. Covert allocators are functions which allocate memory and then return it to the calling function, leaving the ownership of the object slightly ambiguous. As an example of a misunderstood function interface they show some code from Samba which calls a method called trusted_domain_password_delete, and then does not free the argument passed to that method, on the assumption that the method with delete in its name calls delete on its argument. Garbage-collected programs are also vulnerable to memory leaks, but only if references to objects are held longer than required. The leak is not that memory exists which *cannot* be freed, only that memory exists which *is* not freed. Dangling pointers, on the other hand are impossible in garbage-collected languages.

Dangling pointers

If memory is reclaimed too early, memory which is still in use may be overwritten with completely different and inappropriate contents. This is known as a dangling pointer error. It is possible that two different procedures will both be reading and writing to the same piece of memory, but with completely incompatible expectations about the contents of the memory. This can cause unpredictable program behaviour or crashes. These behaviours will be non-reproducible, and therefore very difficult to diagnose. The exact behaviour will depend on when the memory is reallocated and on the patricular timing of the program traces. A dangling pointer is shown in figure 2.1.2.

These errors are some of the most common in languages with explicit memory management. In their study of service reports from an IBM operating system, Sullivan and Chillarege found that 31% of all high-impact errors and 17% of all errors so severe they required a machine reboot were due to memory being freed too early [84].

Detecting dangling pointers is a more difficult problem than detecting memory leaks because static analysis alone cannot detecting dangling pointers except for in certain restricted language subsets [34] or in languages extended with annotations [40]. Even with runtime analysis, it is often necessary to do a source-to-source translation of the program rather than just relinking to an instrumented version of malloc/free [6]. Some of the tools used for memory leak detection can be used for dangling pointer detection, but often with less satisfactory results. Runtime tools introduce a significant time and memory overhead, and may not detect dangling pointers in less commonly executed code paths [33].

2.2 Garbage collection for the competent

While the fact that it reduces development effort and error levels is fairly undisputable, there are many other compelling reasons for using garbage collection unrelated to simple skill requirements or effort levels. Garbage collection can offer architectural advantages over explicit memory management.

Wilson argues that garbage collection is not only desirable but necessary in modular programming, in order to avoid introducing unneeded inter-module dependencies [95]. The crux of his argument is that liveness is a global property, while explicit memory management must be done locally. A routine operating on a data structure should not have to know what other routines are operating on that structure. Before objects can be explicitly deallocated, the module performing the deallocation must be certain that no other modules are interested in that object.

He maintains that this non-local bookkeeping destroys module's properties of orthogonality, composability, and reusability. It hampers the maintainability and extensibility of the code. While many criticisms of garbage collection focus on its peformance impact, Wilson argues that the bookkeeping required for explicit memory management can have an equally significant - but generally unmeasurable - runtime cost.

2.2.1 Managing object ownership

It is a common best practice with manual memory management to require that if "you create it, you free it" or, in the C case, "you malloc it, you free it." [79, 5, 29, 3]. While this advice is inevitably presented as perfectly obvious and simple, it actually becomes complicated to follow it without risking dangling pointers when objects are passed from one component to another.

For example, a reply to a question on a forum about the Python C extensions and when arrays should be released elicited the response (emphasis added) that

> *Unless* some other part of your program holds on to it, of course you have to release it. It's a bit surprising that you have to ask this, really. [61]

Somewhat contradictorily, this advice was followed two sentences later by the confident statement "It's quite simple, really: You malloc it, you free it."

The first complication involves the definition of allocation. Some functions return initialised memory, and these have to be counted as allocation for the pattern to work. The most important of these in C is the strdup function, which duplicates a string. The returned string is created within the strdup function, but it cannot be freed within the strdup function or there would be nothing to return. Therefore strdup is counted as equivalent to malloc.

Even more complicated is the situation where an object is passed to another component, but a reference is also preserved locally. In order to allow object allocators to free objects without creating problems for other components which may hold references to those objects, several elaborate ownership protocols have been devised. Most involve some form of copying objects in order to preserve modularity [57]. When a routine is passed an object which it wishes to hold on to, it will copy the object so that the owner of the original object can deallocate it without risk of a dangling pointer. The second routine takes responsibility for deallocating its copy of the object. More sophisticated protocols also involve some concept of ownership transfer.

The simplest protocol is the C++ orthodox canonical form (OCF) [83]. The orthodox canonical form lays out conditions required to guarantee the success of an object-copying technique.

The OCF states that all classes which manage heap memory structures should provide a default constructor and destructor, a copy constructor, and an assignment operator. The constructor and destructor are fairly obviously required if memory is to be allocated and deallocated. The need for the copy constructor and assignment operator is less obvious and has to do with the need to manage object ownership correctly. The assignment operator is provided so that whenever an object is assigned a local copy is created, and the copy constructor is provided so that the copy created by the assign method is correct.

For example, if a copy constructor is not provided, the default will be used, and an object with references to other objects will be copied deeply instead of shallowly. The referenced objects are now

aliased, with multiple objects holding the same references. When either of the holding objects are freed, the objects it refers to will also be freed. This is unfortunate since another object exists with references to those objects; the references are now dangling pointers. Even in the best case where the dangling objects are never accessed again, any attempt to free the surviving object will involve freeing the destroyed child objects. The memory management system will in general cause a runtime error when free memory is re-freed.

While the OCF safely eliminates inter-module memory dependencies, it incurs a severe performance penalty. If the OCF is strictly followed, many copies are defensively created under conditions when there is no risk of an ownership ambiguity or dangling pointer; every assignment of a non-trivial object results in a copy operation. Simons provide an analysis of several basic operations on an OCF-compliant Set and shows how each one results in multiple unneeded copies being created.

In order to avoid excess copying, more elaborate protocols for ownership transfer have been developed. Simons provides one called larceny, which is a kind of destructive copy in which the copied object loses all its references. This would only be suitable for use when the copied object was about to be destroyed anyway. It is valuable as a workaround for economical assignment since the normal assignment operator forces an expensive copy. Simons does note that larceny should be used carefully and must only be used when variables are clearly temporary. It also requires considerable skill on the part of the programmer to recognise situations in which excess copying might otherwise occur. In other words, larceny eliminates some of the overhead of the OCF pattern, but at the risk of re-introducing memory management errors if care is not taken in programming.

Simons provides a hypothetical performance evaluation of the pilfering pattern relative to OCF code for a small application with many set unions. He first observes that hand-optimisation of the code to re-express the set unions delivered a forty-fold improvement over the version in which the set unions were expressed in the most natural form. He suggests that applying the pilfer pattern would eliminate a further two in three deep copies, giving a non-trivial further improvement. Leaving aside the pilfer pattern for the moment, it is clear that in Simon's example the defensive OCF methods for managing memory and object ownership were tremendously expensive if not optimised correctly.

Even in the more optimal copying protocols, there may still be multiple copies of objects which exist only to prevent memory management errors. This is inelegant in several ways. The extra copying takes time. It can also introduce a significant bloat into the memory requirements of the program.

Object ownership is also awkward for methods which create objects and return them to a caller. Under the 'if you allocate it, you free it' rule, the object-creation code has responsibility for freeing the object. This is clearly impossible when the object must be returned for use elsewhere. The problem is resolved by passing in a pre-allocated but un-filled-in chunk of memory. This awkward pattern is at best inelegant and can be a serious barrier to expressiveness. It breaks encapsulation by requiring the calling function to understand the memory requirements of the created object. Objects can never be a variable size, which may lead to some objects always being wastefully created at their maximum possible size.

Copying is obviously unsuitable for objects using the singleton pattern, and so yet more protocols address the destruction of singletons on program termination [59]. In some cases object ownership is simply too complex for even an elaborate protocol to unravel. In these cases programmers may statically allocate some objects, simply in order to avoid making decisions about deallocating them. These statically allocated objects can impair the scalability of the program.

Ownership protocols can become involved enough that they themselves are a source of errors, as well as performance overhead. Xie's example of the trusted_domain_password_delete method in the Samba codebase is an example of a method whose interpretation of the protocol in use differs from that of the caller. The method operated on the basis that a function never frees its arguments, while the caller assumed that a method with 'delete' in the name would call `delete()` on its argument.

2.2.2 DIY garbage collection

An even more subtle problem caused by the absence of garbage collection is the introduction of garbage collection. Specifically, in large scale programs, the development team often implement some form of application-specific garbage collection. The most popular of these are reference counting on some

data types to avoid the need for each reference to the object to refer to a unique instance. It also common to wrapper the allocation and deallocation routines with ones which track used memory and produce debug information which can be used to check for leaks. Much like the widely available memory leak detection tools, these wrappers may add a performance overhead. The domain-specific checkers are also generally not so heavily optimised as the popular checkers. Although it is relatively easy to implement in a domain-specific way, reference counting is one of the least performant forms of garbage collection [55].

In his garbage collection benchmarking work, Zorn noted that four of the six C programs he tested had implemented domain-specific allocation function for some of the classes, and one of them had implemented a form of reference counting [98].

Wilson argues that these collectors are often buggy and incomplete because they are intended only for use in a particular application. Zorn noted that most of the intended allocation optimisations in the programs he measured were either performance neutral or even worsened performance, and concludes that developer intuition about allocation is often wrong.

Wilson suggests that the continued use of these domain-specific garbage collectors, despite their inadequacy, is proof of the value of garbage collection, and of the need for garbage collection as a core part of a programming language implementation. The suboptimal program structure caused by the absence of garbage collection make performance comparison between garbage collected and non-garbage collected languages difficult. The performance impact of garbage collection will be discussed in more detail in chapter 7.

Garbage collection offers many architectural and software engineering advantages which can result in more elegant and efficient code. It also offers a number of runtime advantages unrelated to the collection of garbage; these are discussed in chapter 3.

Chapter 3

Garbage collection is all about collecting garbage

In Beverly Hills [. . .] they dont throw their garbage away. They make it into television shows.

Woody Allen, *Annie Hall*

The term 'garbage collection' is often used as a shorthand for automatic memory management. However, automatic memory management involves much more than just the collection of disused objects from the heap, or strictly literal 'garbage collection'. In particular, memory management also implies memory allocation. Appropriate memory management can make allocation more efficient by greatly reducing fragmentation and contention. Memory management can also be used to optimise performance by rearranging objects to increase cache efficiency.

3.1 Garbage collection and the heap

Automatic memory management mediates the interaction between applications and the heap.

Applications may read objects on the heap, write objects on the heap, allocate new objects, free objects, and perform calculations. In simplified form, the set of application actions may be represented as

$$MutatorActions = \{|readobject, writeobject, allocateobject, freeobject, calculate|\}$$

$$Mutator = \sqcap x : MutatorActions \bullet Mutator$$

An application consists of one or more mutator threads interleaved with one another. (Intra-application synchronisation is not relevant to this model.)

$$Application = |||\, x : 1 \,..\, m Mutator$$

Mutators require the cooperation of a processor.

$$Processor = \Box\, x : MutatorActions \bullet Processor$$

Access to memory requires a heap.

$$HeapActions = \{|readobject, writeobject, allocateobject, freeobject|\}$$

All but the calculations require the cooperation of the heap.

$$MutatorActions \setminus HeapActions = \{|calculate|\}$$

The process of an N-processor machine consists of the processors interleaved with one another, all synchronised with the heap.

$$Machine = \left(\|\| n : 1..N \bullet Processor\right) [| \alpha(Heap) |] Heap$$

The total system is then composed of the machine synchronised with the application.

$$System = Machine [| ApplicationActions |] Application$$

Heaps allow objects to be read, written to, allocated, freed, and moved. When an object is freed it sometimes does not increase the amount of free space in the heap, because of fragmentation.

$$\begin{aligned} Heap(n) = {} & readobject?o \rightarrow Heap(n) \\ & \Box \\ & writeobject?o \rightarrow Heap(n) \\ & \Box \\ & n < N \;\&\; allocateobject!o \rightarrow Heap(n+1) \\ & \Box \\ & n > 0 \;\&\; freeobject?o \rightarrow Heap(n-1) \\ & \Box \\ & n > 0 \;\&\; freeobject?o \rightarrow Heap(n) \\ & \Box \\ & n > 0 \;\&\; freeobject?o \rightarrow Heap(n) \\ & \Box \\ & n > 0 \;\&\; moveobject?o \rightarrow Heap(n) \end{aligned}$$

The garbage collector participates in both the freeing and allocation of objects. (As will be discussed in chapter 5, some may also participate in the reading and writing of objects.)

$$\begin{aligned} AbstractGarbageCollector = {} & allocateobject?o \rightarrow AbstractGarbageCollector \\ & \sqcap \\ & freeobject?o \rightarrow AbstractGarbageCollector \\ & \sqcap \\ & moveobject?o \rightarrow AbstractGarbageCollector \end{aligned}$$

In the garbage collected system, applications do not free or move objects. This is expressed by synchronising the *freeobject* and *moveobject* events with the *Stop* process.

$$\begin{aligned} AbstractGCSystem = {} & Machine \\ & [|ApplicationActions|] (Application [| \{|freeobject, moveobject|\} |] Stop) \\ & [|\{|freeobject, allocateobject, moveobject|\}|] AbstractGarbageCollector \end{aligned}$$

Garbage collectors tend to make their involvement in freeing objects explicit by starting and stopping well-defined collection phases.

$$\begin{aligned} VerboseCollector = {} & allocateobject?o \rightarrow VerboseCollector \\ & \sqcap \\ & allocateobject?o \rightarrow begincollection \rightarrow CollectingVerboseCollector \end{aligned}$$

$$\begin{aligned} CollectingVerboseCollector = {} & freeobject?o \rightarrow CollectingVerboseCollector \\ & \sqcap \\ & freeobject?o \rightarrow endcollection \rightarrow VerboseCollector \\ & \sqcap \\ & moveobject?o \rightarrow CollectingVerboseCollector \\ & \sqcap \\ & moveobject?o \rightarrow endcollection \rightarrow VerboseCollector \end{aligned}$$

The garbage collectors may be joined with a verbose gc logger to report on these events.

$$\begin{aligned} VerboseGCMonitor(n) = {} & tock \rightarrow VerboseGCMonitor(n+1) \\ & \Box \\ & endcollection \rightarrow report!n \rightarrow IdleVerboseGCMonitor \end{aligned}$$

$$\begin{aligned} IdleVerboseGCMonitor = {} & tock \rightarrow IdleVerboseGCMonitor \\ & \Box \\ & startcollection \rightarrow VerboseGCMonitor(0) \end{aligned}$$

The verbose gc monitor uses a *tock* model of time.

$$LoggedCollector = VerboseCollector \; [|\; \{|startcollection, endcollection|\} \;|] \; IdleVerboseGCMonitor$$

Because the verbose monitor reports time spent during the collection phase, it is easy to neglect the impact the collector's allocation activity may have on the application.

3.1.1 Caches and locality

All modern computer systems use caches to speed up access to the heap. Heap access is very slow relative to instruction processing. In order to prevent memory access from being a serious performance bottleneck, memory caches are introduced. The caches are smaller than the heap but access to them is much faster. It is common to include several layers of caching with decreasing size and increasing speed. This model only includes one layer. When an object is accessed in memory, an attempt is first made to retrieve it from the cache. If it is not present in the cache, it (along with its neighbours in memory) is loaded into the cache, and it is then read from the cache.

$$\begin{aligned} Cache(c) = {} & o \in c \;\&\; readobject?o \rightarrow Cache(c) \\ & \Box \\ & o \in c \;\&\; writeobject?o \rightarrow Cache(c) \\ & \Box \\ & o \in c \;\&\; allocateobject?o \rightarrow Cache(c) \\ & \Box \\ & o \in c \;\&\; freeobject?o \rightarrow Cache(c) \\ & \Box \\ & o \notin c \;\&\; loadtocache?o!c?d \rightarrow readobject?o \rightarrow Cache(d) \\ & \Box \\ & o \notin c \;\&\; loadtocache?o!c?d \rightarrow writeobject?o \rightarrow Cache(d) \\ & \Box \\ & o \notin c \;\&\; loadtocache?o!c?d \rightarrow allocateobject?o \rightarrow Cache(d) \\ & \Box \\ & o \notin c \;\&\; loadtocache!o!c?d \rightarrow freeobject?o \rightarrow Cache(d) \\ & \Box \end{aligned}$$

where the following invariants hold for the $loadtocache!o!c?d$ event:

$$\begin{aligned} & \#\, c = \#\, d \\ & o \in d \end{aligned}$$

The *loadtocache* events are expensive in terms of time, and avoiding them is desirable. Locality is the effect of physical memory location on the speed of object access. Objects that are stored spatially close to recently accessed objects can be accessed very quickly they will also have been loaded into the cache. The unit of memory which is loaded at the same is called a cache line. As improvements in processor speed outpace improvements in cache and memory speed, locality is becoming increasingly important to the performance of modern applications [53].

3.2 Allocation

Even systems without garbage collection require some way of handling object allocation and freeing. Usually this takes the form of a library, sometimes integrated with the operating system, and sometimes a discrete component. For example, in C, allocation is handled by the `malloc/free` library.

$$\begin{aligned} \mathit{MallocFreeLibrary} = {} & \mathit{allocateobject?o} \rightarrow \mathit{MallocFreeLibrary} \\ & \Box \\ & \mathit{freeobject?o} \rightarrow \mathit{MallocFreeLibrary} \end{aligned}$$

Conventional allocator design is non-trivial and has been the subject of much research [96], and there exist many implementations of malloc, including ptmalloc, mtmalloc, lkmalloc, phkmalloc, dmalloc, and jemalloc. Some are optimised for single-threaded systems, others for multi-threaded systems, some for small objects, and others for large objects. For certain applications, the choice of allocator can have a significant impact on performance [31]. In the most extreme cases, the difference between a high-performing and low-performing allocator is a factor equal to the number of processors on the box [15]. In these multi-threaded application cases, the allocator can be a serious bottleneck for performance if chosen unwisely.

While not all garbage collectors include allocation components, the vast majority do. Even collectors which operate in hostile environments such as the Boehm-Demers-Weiser collector for C and C++ [19] includes an allocator. For other collectors which are more integrated into the programming language, allocation is an integral part of the garbage collection algorithm.

Factors which affect the performance of allocators include contention between threads, locality of reference, cache behaviour, and fragmentation. Some of these factors affect the time taken to perform the allocation, some affect the application performance, and some affect both.

3.2.1 Allocation contention and thread local heaps

In a multithreaded system, many mutator threads will be synchronised with a single heap. This synchronisation can hamper application activity as many threads contend for locks on a single heap. If every thread wishes to perform an *allocateobject* event and the heap can only service one thread at a time, thread execution will be delayed.

Many modern allocation libraries and almost all modern garbage collectors solve this problem by batch-allocating memory. When a thread attempts to allocate an object it receives a chunk of memory, known as a thread-local heap. Further allocation requests are serviced by this mini-heap until it is exhausted. The thread-local heap synchronises exclusively with the originating thread, reducing contention.

Contention can also be critical once the objects are laid out. On multi-threaded systems, if two processors are manipulating data on the same cache line, they will have to arbitrate ownership of that cache line [15]. This is known as false sharing and seriously degrades performance. It is therefore a good idea to try and ensure that data associated with different processors does not end up in the same cache line.

3.2.2 Fragmentation

Allocation from a fragmented heap is significantly more costly than allocation from an unfragmented heap. When the heap is fragmented, allocation involves potentially lengthy searches through free lists. In the worst case, the heap may be so fragmented that allocation is impossible. In completely unfragmented heaps, on the other hand, allocation simply required a pointer be bumped, and there is no free list search at all. Copying collectors allow this kind of super-efficient allocation. Sun report that the typical path for allocating a new object in their generational collector is only ten native instructions [26].

With a free list heap, all object allocations require that the free list be searched. There is a possibility that even after searching the free list, in a heap with sufficient capacity, allocating the

object is impossible. Freeing objects also requires that the free list be updated to include the defunct object.

$$
\begin{aligned}
\mathit{FreeListHeap}(n, f) = {} & \mathit{readobject}?o \rightarrow \mathit{FreeListHeap}(n, f) \\
& \Box \\
& \mathit{writeobject}?o \rightarrow \mathit{FreeListHeap}(n, f) \\
& \Box \\
& n < N \ \& \ \mathit{searchfreelist}?o!f?g \rightarrow \mathit{allocateobject}!o \rightarrow \mathit{FreeListHeap}(n+1, g) \\
& \sqcap \\
& n < N \ \& \ \mathit{searchfreelist}?o!f?g \rightarrow \mathit{FreeListHeap}(n, g) \\
& \Box \\
& n > 0 \ \& \ \mathit{freeobject}?o \rightarrow \mathit{updatefreelist}?o!f?g \rightarrow \mathit{FreeListHeap}(n-1, g)
\end{aligned}
$$

With a compacted or copying heap, no costly free list search is required. The distinction between the two is that the copying heap will stay a copying heap, while the compacted heap will at some stage become an uncompacted heap once it becomes sufficiently fragmented. The other more obvious distinction is that a compacted heap still requires a free list, it is just that operations to it are efficient.

$$
\begin{aligned}
\mathit{CompactHeap}(n, f) = {} & \mathit{readobject}?o \rightarrow \mathit{CompactHeap}(n, f) \\
& \Box \\
& \mathit{writeobject}?o \rightarrow \mathit{CompactHeap}(n, f) \\
& \Box \\
& n < N \ \& \ \mathit{fastsearchfreelist}?o!f?g \rightarrow \mathit{allocateobject}!o \rightarrow \mathit{CompactHeap}(n+1, g) \\
& \sqcap \\
& n < N \ \& \ \mathit{fastsearchfreelist}?o!f?g \rightarrow \mathit{allocateobject}!o \rightarrow \mathit{FreeListHeap}(n+1, g) \\
& \Box \\
& n > 0 \ \& \ \mathit{freeobject}?o \rightarrow \mathit{fastupdatefreelist}?o!f?g \rightarrow \mathit{CompactHeap}(n-1, g)
\end{aligned}
$$

A semispace (copying) heap will never refuse allocation if it has capacity, because it will always be perfectly compacted. The downside is that objects cannot be freed on an ad-hoc basis; the entire heap must be cleared in a batch.

$$
\begin{aligned}
\mathit{SemispaceHeap}(n, O) = {} & \mathit{readobject}?o \rightarrow \mathit{SemispaceHeap}(n, O) \\
& \Box \\
& \mathit{writeobject}?o \rightarrow \mathit{SemispaceHeap}(n, O) \\
& \Box \\
& n < N \ \& \ \mathit{allocateobject}!o \rightarrow \mathit{SemispaceHeap}(n+1, O \cup o) \\
& \Box \\
& n > 0 \ \& \ (\fatsemi \ o : O \bullet \mathit{freeobject}?o \sqcap \mathit{moveobject}?o) \rightarrow \mathit{SemispaceHeap}(m, M)
\end{aligned}
$$

In this case freeing an object simply means not copying it. Allocation is very fast because it simply means incrementing the high water-mark pointer.

In order to achieve this kind of performance objects must be re-arranged to fill in holes as objects die, which is generally impossible without garbage collection. Detlets et al. tested four implementations of `malloc/free`, with eleven different applications, the mean allocation overheads ranged from 52 and 92 instructions. This is five to nine times greater than the costs reported by Sun for the copying collector.

It must be noted that this low allocation cost is certainly not a fundamental property of a garbage collected system. Detlefs et al. also report the allocation overhead of version 2.6 of the Boehm-Demers-Weiser conservative garbage collector. For the Boehm-Demers-Weiser collector, the mean allocation overhead was 611 instructions. For the Ild incremental loader, which had unusually large average object sizes, the allocation overhead was an eye-watering 3544 instructions per call. Of course, the Boehm-Demers-Weiser collector is operating without language support or opaque pointers, and so it cannot rearrange objects to reduce fragmentation. It maintains multiple free lists for different objext sizes and also different object types [20].

In fact, the poor allocation performance, rather than the performance impact of the collection phases, may be responsible for the performance difference between the BDW collector and the conventional malloc/free libraries. Over the eleven tested applications, the performance of the BDW collector was 21% worse than that of the best-performing library. Detlefs et al. do not report how many instructions the BDW collector spent garbage collecting, but it spent between six and eleven times as long allocating each object as the other allocators.

It is also worth noting that some modern C allocators are able to offer better performance than those measured by Detlefs et al with more sophisticated free-list algorithms and fragmentation-reducing `free` operations. For example, the Hoard collector requires only a "handful" of instructions to allocate [15].

Allocation from less-fragmented heaps is also likely to lead to improved application performance in general. Objects which are allocated close together in time are likely to be accessed close together in the future. If these objects are allocated from an unfragmented heap, they will be spatially close together as well and likely to occupy the same cache line.

3.3 Object re-ordering

In combination with runtime profiling (such as that obtained from a Just-In-Time compiler), a garbage collector can reorder objects into an optimum order. Objects which tend to be accessed close together in time can be positioned so that they are physically close to each other in memory, and likely to be located on the same cache line. This can reduce cache misses and improve mutator performance. Huang et. al were able to obtain mutator time performance improvements between a few percent and 45% across a suite of SPECjvm and DaCapo benchmarks by changes to object order. They used the JIT to profile hot methods, and also a profile of field accesses within a method obtained by instrumentation. They combined these two pieces of information to produce a list of hot fields, and grouped these together when copying.

Shuf et al. reported throughput improvements of 21% by adjusting the Java object allocation algorithm [81]. They identified certain popular object types as prolific. When these objects were allocated, space was reserved next to them for their children. That is, the free-list pointer was moved further forward than necessary in order to ensure colocation of parents and children. Since parents and children are often accessed contemporaneously, this improves the mutator cache performance. They also argue it speeds future allocation performance by decreasing memory fragmentation. Since objects that are created together and refer to one another tend to die together [52], locating objects which refer to one another close together means larger contiguous chunks of memory are likely to be freed by the garbage collector. (If the predicted child object is never actually allocated, of course, this allocation technique increases memory fragmentation.)

These results are presented as illustrative only, since allocator performance is extremely sensitive to application allocation patterns.

Chapter 4

Small pause times lead to good response times

As opposed to the incoherent spectacle of the world, the real is what is expected, what is obtained and what is discovered by our own movement. It is what is sensed as being within our own power and always responsive to our action.

Alain Chartier

In applications where response times are critical, it is common to choose a concurrent collection mode and accept the reduced application throughput. However, the reduced throughput can sometimes lead to larger mean response times, despite the shorter interruptions in application activity. This apparent paradox can be understood with a queueing theory analysis of garbage collection response times and their relationship with throughput and pause times.

4.1 Real-time systems

The myth that small pause times give good response times is present even in the academic literature. Response times are of most concern for real-time systems. Real-time systems are those with strict requirements on the responsiveness on the system. Computational delays are not acceptable, usually because the consequences of the delays would be serious. For example, systems controlling objects travelling at high speeds, such as planes, cars, or missiles, must be able to respond to environmental events extremely quickly.

While garbage collection is generally considered incompatible with real-time requirements, real-time-compliant garbage collection has been an active research area. Most of the research has focussed almost exclusively on reducing pause times [11, 12].

However, Cheng and Blelloch point out that small pause times are no guarantee of small response times if the pauses are closely spaced [24]. They argue that multiple small pauses are not really that different from one long pause. In order to get more usefully predict responses time based on pause times, they define a property called the minimum mutator utilisation (MMU). The MMU is the smallest proportion of time devoted to the mutator over all intervals of size Δt in the course of the application run. As Δt approaches the length of the application run, the MMU $u(\Delta t)$ approaches $1 - g$, where g is the collector's pause time overhead. For smaller intervals the MMU will depend significantly on characteristics of the collector. Bacon argues that the MMU is much more likely to be a good predictor of worst-case response times and that many attempts at real time garbage collection are fundamentally flawed because of their emphasis on pause times:

> Baker attempts to finesse this problem by interleaving the collector with the mutator in a very fine-grained manner, but this only hides the problem: it keeps individual pauses low, but does not prevent numerous closely-spaced pauses.

In the case of real-time systems, small pause times must not be assumed to ensure low response times, because if the pauses are very closely spaced, the mutator will be starved of the CPU time it requires to respond quickly.

4.2 Concurrent collection

A similar problem exists when collections are performed mostly-concurrently. In this case the inhibitor to speedy response-times is the collection load placed on the mutator, rather than the frequency of collection.

4.2.1 The SPECjbb benchmarks

SPECjbb represents a three-tier transaction system. User interaction is simulated by random input selection and the third tier, the database, is represented by a set of binary trees. The primary focus of the measurement is the middle tier which handles the business logic.

SPECjbb was designed to test several properties of a JRE. It is designed to test how well systems scale as the number of active threads increases, but the active data set also increases linearly with the number of threads [56]. This means garbage collection variations can have a significant effect on the measured scalability of the system.

The first tier is a collection of clients. Clients place orders and check order status.

$$ClientEvents = \{| placeorder, checkstatus |\}$$

The client may perform these events, or it may perform other generic processing to prepare itself for these events:

$$\begin{aligned} Client = {} & (\sqcap\, a : ApplicationActions \bullet Client) \\ & \sqcap \\ & (\sqcap\, a : ClientEvents \bullet Client) \end{aligned}$$

The bulk of the work is done by warehouses, which cooperate in the placing of orders and the checking of status, and which check stock levels, process orders for delivery, enter customer payments, and request customer reports in order to do these.

$$WarehouseEvents = \{| processorder, enterpayment, checkstocklevel, requestcustomerreport |\}$$

The warehouse also makes queries of the back-end database.

$$DatabaseQueries = \{| query |\}$$

$$\begin{aligned} Warehouse = {} & (\sqcap\, a : ApplicationActions \bullet Warehouse) \\ & \sqcap \\ & (\sqcap\, c : WarehouseEvents \bullet Warehouse) \\ & \sqcap \\ & (\sqcap\, d : DatabaseQueries \bullet Warehouse) \end{aligned}$$

Each warehouse is associated with a single client:

$$Thread = Client \;[|\; ClientEvents \cup WarehouseEvents \;|]\; Warehouse$$

A single database serves all warehouses:

$$\begin{aligned} Database = {} & (\sqcap\, a : ApplicationActions \bullet Database) \\ & \sqcap \\ & (\sqcap\, d : DatabaseQueries \bullet Database) \end{aligned}$$

The company consists of a number of client–warehouse pairs, along with the master database:

$$\begin{aligned} Company(n) = ((\|\!|\!|\, x : \{1..n\} \bullet Thread) \,[|\, DatabaseActions \,|]\, Database) \\ \|\, periodcompleted \rightarrow Skip \end{aligned}$$

Since it is intended to measure system performance, time (represented by *tock*s in this model) is obviously important within SPECjbb. The main performance metric is the throughput, subject to a minimum response time criterion. Throughput is defined as the number of transactions completed, where

$$Transactions = ClientEvents \cup WarehouseEvents$$

The response time criteria is enforced with a response time monitor as

$$\begin{aligned} ResponseMonitor(t) = {} & tock \rightarrow ResponseMonitor(t+1) \\ & \Box \\ & n \leq T \;\&\; \Box\, e : Transactions \bullet counttransaction \rightarrow ResponseMonitor(0) \\ & \Box \\ & n > T \;\&\; \Box\, e : Transactions \bullet ResponseMonitor(0) \end{aligned}$$

where T is the maximum response time.

$$\begin{aligned} ThroughputMonitor(n) = {} & tock \rightarrow ThroughputMonitor(n) \\ & \Box \\ & \Box\, e : Transactions \bullet (counttransaction \rightarrow ThroughputMonitor(n+1) \\ & \qquad\qquad \Box \\ & \qquad\qquad ThroughputMonitor(n)) \\ & \Box \\ & periodcompleted \rightarrow reportthroughput!n \rightarrow ThroughputMonitor(0) \end{aligned}$$

$$MeasurementPeriod(n) = System(n) \,[|\, Transactions \,|]\, ThroughputMonitor$$

Finally, each measurement period has a fixed duration, enforced with a timer,

$$\begin{aligned} PeriodTimer(n,t) = {} & t < P \;\&\; tock \rightarrow PeriodTimer(n, t+1) \\ & \Box \\ & t = P \wedge n < N \;\&\; periodcompleted \rightarrow PeriodTimer(n+1, t) \\ & \Box \\ & n = N \;\&\; periodcompleted \rightarrow Skip \end{aligned}$$

where N is the maximum number of warehouses.

A SPECjbb run consists of a series of measurement periods, with the number of warehouses increasing by one between each period. There are two major versions of the benchmark, with the first being SPECjbb2000:

$$\begin{aligned} SPECjbb2000 = {} & (;\, x : \{1..N\} \bullet forcedgc \rightarrow MeasurementPeriod(x)) \\ & [|periodcompleted|] \\ & PeriodTimer(1,0) \end{aligned}$$

SPECjbb2005 was refactored and updated in several ways, including one change which was particularly significant for garbage collection. The original benchmark did a forced garbage collection between every measurement period. The time taken for this collection was not included in any of the benchmark metrics. In other words, the forced collection is 'free'. This means that garbage collectors which created severe fragmentation but compacted during forced collections could produce very good SPECjbb scores, despite their unsuitability for any long-running server application. Another strategy which produces even less representative results is to use a heap so large that garbage collections are

avoided entirely during a measurement period, and then collect the massive heap without penalty between measurement periods.

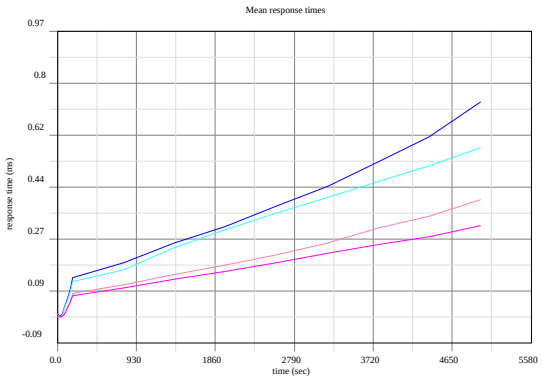
$$SPECjbb2005 = (\fatsemi\ x : \{1..N\} \bullet MeasurementPeriod(x)) \ [|\ periodcompleted\ |]\ PeriodTimer(1,0)$$

Because SPECjbb2005 eliminates the free forced collection [28], many JREs or option combinations which produced good results on SPECjbb2000 produce poor results on SPECjbb2005.

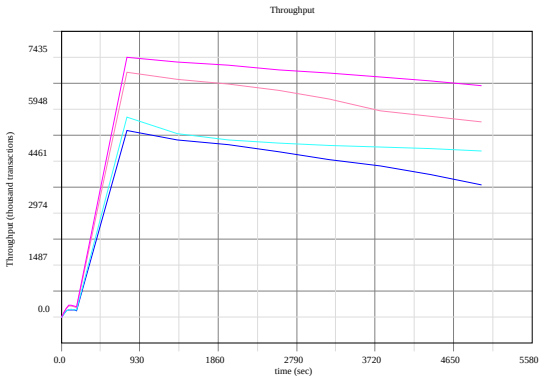
4.2.2 Response times and pause times

SPECjbb can give insight into the effects of garbage collector changes on both throughput and response times. While the canonical score is largely based on the throughput, response times are also reported in the raw log files.

An example set of mean SPECjbb response times for several system configurations are shown in figure 4.1(a). The response times are reported per warehouse. The response times increase steadily with the load on the system. Similarly, the throughput, shown in figure 4.1(b), declines as the load increases once the number of threads exceeds the number of processors in the machine.



(a) Mean response times for the SPECjbb 2005 benchmark and several garbage collection policies.



(b) Throughput for the SPECjbb 2005 benchmark and several garbage collection policies.

Figure 4.1: Mean response times and throughput for SPECjbb2005 run with several system configurations. The magenta and salmon lines were obtained using a mark-sweep policy, while the blue and cyan lines used a concurrent collection policy. The heap size was fixed at 2000 MB for the magenta and cyan lines, and 1200 MB for the salmon and blue lines. The mean response time and throughput are well correlated.

Policy	Heap (MB)	Mean pause time (ms)	Mean response time (ms)	Mean throughput (ktransactions)
Concurrent	1200	14	0.29	2592
	2000	19	0.25	2899
Mark-sweep	1200	463	0.16	3598
	2000	481	0.14	4036

Table 4.1: Pause times, response times, and throughput for concurrent and non-concurrent garbage collection policies.

The pause times for the same SPECjbb run are shown in figure 4.2 and summarised in table 4.1. In general the pause times for the mark-sweep policy are large and correlated to the amount of live data, and the pause times for the concurrent policy are small and relatively constant. The spikes

in the concurrent pause times represent collections where the concurrent collection did not complete before memory ran out, and so part of the collection had to be redone non-concurrently.

Figure 4.2: Garbage collection pause times for the SPECjbb 2005 benchmark and several garbage collection policies. The magenta and salmon lines were obtained using a mark-sweep policy, while the blue and cyan lines used a concurrent collection policy. The heap size was fixed at 2000 MB for the magenta and cyan lines, and 1200 MB for the salmon and blue lines.

Even maximum pause time is poorly correlated to maximum response time. The maximum response time is at least as big as the maximum pause time, unsurprisingly. However, the dominant factor in the response times for the poorly performing policies does not seem to be the maximum pause time. The maximum pause and response times are shown in table 4.2.2. The largest maximum response time occurred with the mark-sweep policy, because of a compaction in the last few seconds of the run. The mark-sweep policy also boasts the smallest maximum response time, in the run with the smaller heap. The maximum response times for the concurrent policy are well over 4 times the maximum pause time, while the ratio is well under 2 times for the mark-sweep policy.

Policy	Heap (MB)	Maximum pause time (ms)	Maximum response time (ms)
Concurrent	1200	298	1437
	2000	281	1438
Mark-sweep	1200	2107	2252
	2000	638	1099

Table 4.2: The relationship between maximum response times and maximum pause times.

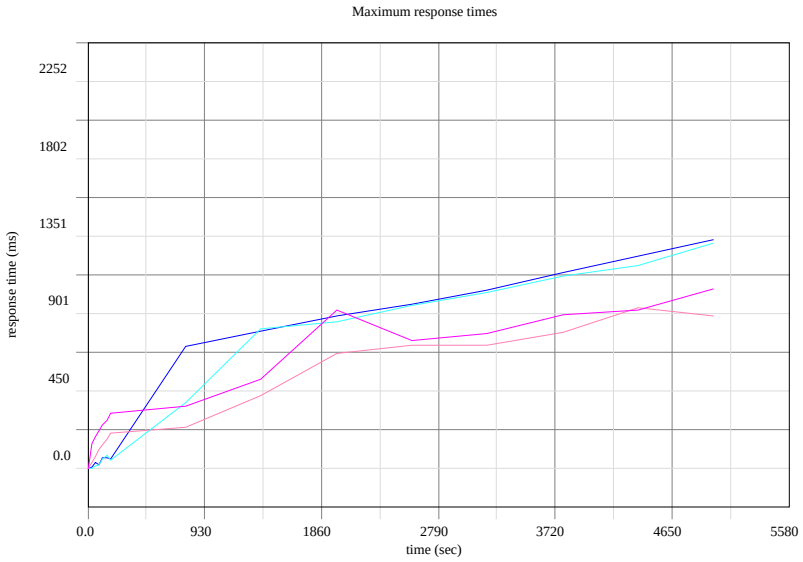


Figure 4.3: Maximum response times for the SPECjbb 2005 benchmark and two garbage collection policies. The maxima are reported per warehouse. The legend is the same as that of figure 4.1.

4.3 Queueing theory model

Figures 4.3 and 4.1(b) were very suggestive that throughput is a good predictor of response times. They also suggested that garbage collection pauses were much less useful as a predictor. Is it possible to confirm such relationships mathematically? Queuing theory concerns itself with is the response times and throughput of a system, and so provides a good framework for a mathematical exploration. In general, a queue is a system in which there are customers arriving for service, customers waiting for service, customers being served, and customers leaving the system after being served. Queueing theory is a powerful and flexible tool. As well as program execution, queueing theory has been used to model a diverse range of problems, such as road traffic, network traffic, service queues in shops, mining transportations, fleet operations, and industrial processes [43].

At the core of queueing theory – unsurprisingly – is the concept of the queue. Figure 4.4 shows an abstraction of a simple queue, modelling a first approximation of program execution. In this case, the customers are application threads, and service is the execution by the processor of an instruction. Instructions arrive from the left, wait in a queue, and are then executed by the processor.

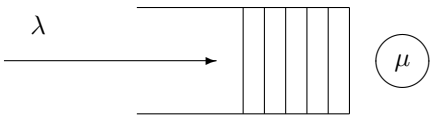


Figure 4.4: A simple queue.

In the simplest model, both the arrival times and service (execution) times are exponentially distributed around some mean. The mean arrival rate is λ instructions/sec, and the service rate is μ instructions/s. In queueing theory notation, this is known as an M/M/1 queue, where the first two 'M's represent exponential (Markov) arrival and service processes, and the '1' represents the number of servers. Figure 4.5 shows the state transition diagram of a simple M/M/1 queue. This is a graphical representation of a Markov chain. Each node represents a state of the system whose defining characteristic is the number of customers present. As customers arrive with rate λ, there is

a state transition to the right. As they are served at rate μ, there is a transition to the left. The left-most state is the state with no customers in the system.

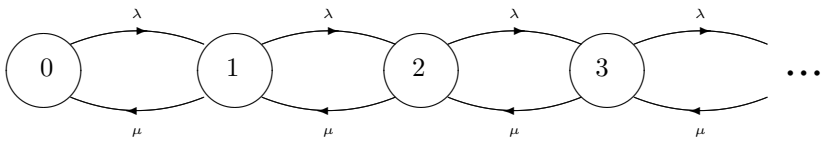


Figure 4.5: The state transition diagram for a simple M/M/1 queue.

The throughput, γ, of the system is the rate at which jobs are processed. It is conventional to require that $\mu < \lambda$, because the queue will grow without bounds and the system will never achieve a steady state if $\mu \geq \lambda$. In the simplest M/M/1 queue, it follows from this restriction that $\gamma = \lambda$. In more complex queues, it may be that $\gamma < \lambda$ if some arriving customers are prevented from entering the queue.

Multi-processor or multi-core systems could be similarly modelled as M/M/n queues. The state transition diagram for these queues is shown in figure 4.6.

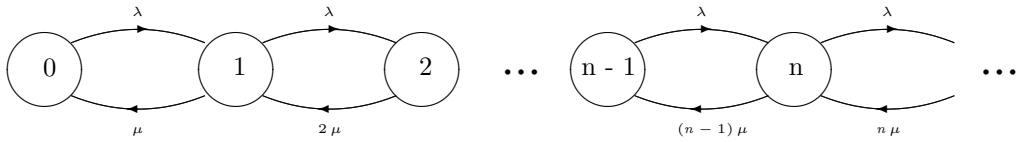


Figure 4.6: The state transition diagram for a simple M/M/n queue.

Probability

Analysis of the queueing model requires some basic concepts from probability. Probability theory involves two important constructs, random variable and events. Events are changes in random variables. The mean of a random variable is more properly known as the expected value, since it describes the mean of a series of situations which have not yet happened. The expected value of a variable V is written $E[V]$. It is often useful to characterise the distribution of a random variable more precisely than just the mean. A common extra characterisation is the variance, which indicates how widely values of the variable deviate from the mean. The variance is defined as

$$V[X] = E[X^2] - E[X]^2 \tag{4.1}$$

The term $E[X^2]$ is known as the second moment of X, while the first moment is simply $E[X]$. In general the nth moment is defined as $E[X^n]$. The higher order moments more precisely characterise the distribution of the random variable.

4.3.1 Instruction-level queueing

Gross and Harris describe six characteristics of a queueing process which should be included in any model [43]: the arrival pattern of customers, the service pattern of servers, the queue discipline, the system capacity, the number of service channels, and the number of service stages. Some of these are more straightforward than others to model.

Arrival pattern of customers

To describe the arrival pattern of the customers, it is necessary to know whether arrivals are determininistic or stochastic, what the probability distribution of interarrival times is in the stochastic case, whether customers can arrive simultaneously, and whether customers balk, renege, or jockey between queues.

In the simplest model of a computer system, instructions are executed sequentially. No instruction requests service until the instruction preceding it has been executed. In this case the service and arrival rates are identical. Such a system is shown in figure 4.7.

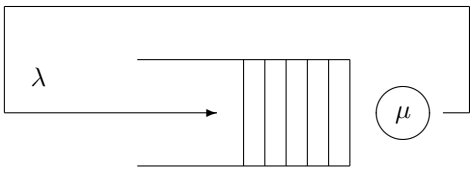


Figure 4.7: A simple cyclic queue, with one processor.

The population of such a system is static and there are no state transitions. Its trivial Markov chain is shown in figure 4.8.

Figure 4.8: The Markov chain for a system with no state transitions.

In a slightly more sophisticated model, software threading allows for the possibility of multiple instructions queueing for service simultaneously. Such threads would spawn at some rate α and terminate with rate β. Such a system is shown in figure 4.9.

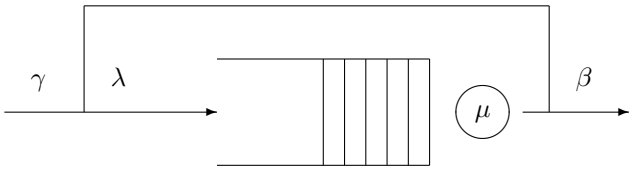


Figure 4.9: A cyclic queue with one processor and software threading.

For a steady state to exist, it must hold that $\gamma = \beta$, that is, that the number of customers entering the system is equal to the number exiting. The Markov chain for this system is shown in figure 4.10. The population of the system increases with rate γ, as new customers enter the queue, and decreases with rate $\mu - \lambda$, the rate of customers who leave the system and do not cycle back.

Customers are unlikely to arrive in clumps, since instructions are executed sequentially. While multiple threads may be active, their interrelationships are likely to be stochastic.

Application threads are unlikely to balk or renege. Balking is particularly unlikely since application threads will generally not have access to the system load statistics (which could indicate queue lengths). Reneging is slightly more likely, either because threads time out in times of very heavy load, or due to system interrupts. Even this, though, is unlikely enough that it can safely be omitted from the model.

Whether application threads jockey for position between different processors will depend on the algorithms used by the operating system for CPU load balancing. For simplicity we assume that there is no particular load balancing and that no jockeying occurs. Load balancing is discussed in more detail in section 4.3.1.

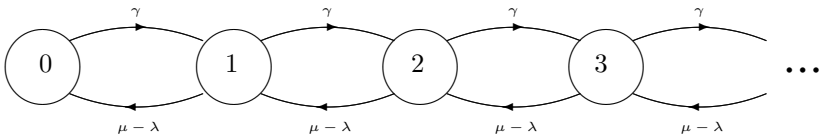


Figure 4.10: The state transition diagram for the queueing system shown in figure 4.9.

Service pattern of servers

The same parameters used to describe the arrival pattern of customers may be used to describe the service pattern of servers. The most important characteristic is the probability distribution in service times, with batching and state-dependency of service times also playing a role.

Several factors determine how long a processor takes to handle an instruction. The first is the instruction, since some instructions will take longer to process than others, particularly in non-RISC systems. A second factor is the state of the memory caches of the system. Instructions which require only access to memory stored in L3 caches will execute quickly. Cache misses requiring access to L2 or conventional RAM memory will cause slower service.

The service pattern of the servers is not state dependent. The CPU will process instructions at the same rate no matter how long the queue of application threads is. (The only exception to this is that more threads in the queue may correspond to more cache pollution which would increase service times. This would be the case no matter how long each thread was queueing, however.)

Queue discipline

Queue discipline describes how customers are selected for service. It could be first-come-first-served, last-come-first-served, random, or prioritised. We assume the instruction execution is first-come-first-served. When the queue with simple feedback shown in figure 4.9 employs a first-come-first-served queue discipline, it is known as a Bernoulli queue [87].

While the system as a whole may have a non-preemptive priority system (at the instruction level) this may not be the case for the application threads. We assume for simplicity that there is no priority system for the application threads.

Number of service channels

The number of service channels in the model maps relatively straightforwardly to the number of cores or CPUs in the system. Simultaneous multi-threading (such as Intel's hyper-threading) [86] is a complication which we will ignore in the model.

In general, when the number of service channels is greater than one, there are two system configurations. In the first, several servers share a single queue. In the second, each server has its own queue. When there is feedback in the system there are two extra permutations, since arrangements may be different for customers coming and by feedback and new customers. A system in which all queues are shared is shown in figure 4.11.

In general modern multi-processor architectures implement some sort of processor affinity. Schemes vary, with some load balancers redirecting existing processes to under-utilised processors (weak affinity or jockeying), and others assigning new processes based on where similar processes have already run (strong affinity). A system which implements something roughly equivalent to strong affinity is shown in figure 4.12; running threads are directed to the queue of the server handling that thread, and new processes are also arbitrarily directed to a server, which may not be the most lightly-loaded one.

Broadly speaking, the most efficient schemes direct new processes to lightly loaded processors, and keep existing processes with their current processor. This corresponds to shared queues for new processes, and individual queues for existing fed-back processes. This starts to strain the queue visualisation notation, but an attempt at drawing such a system is made in figure 4.13. The fourth

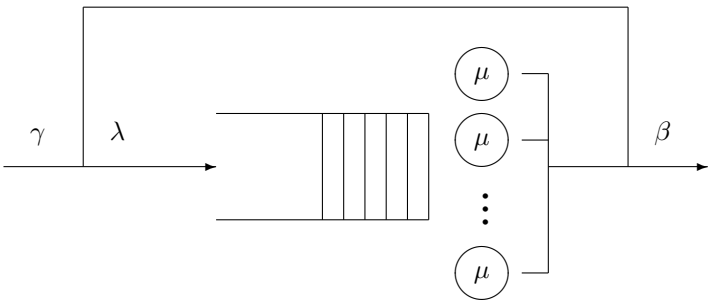


Figure 4.11: A system with feedback and several servers sharing a single queue.

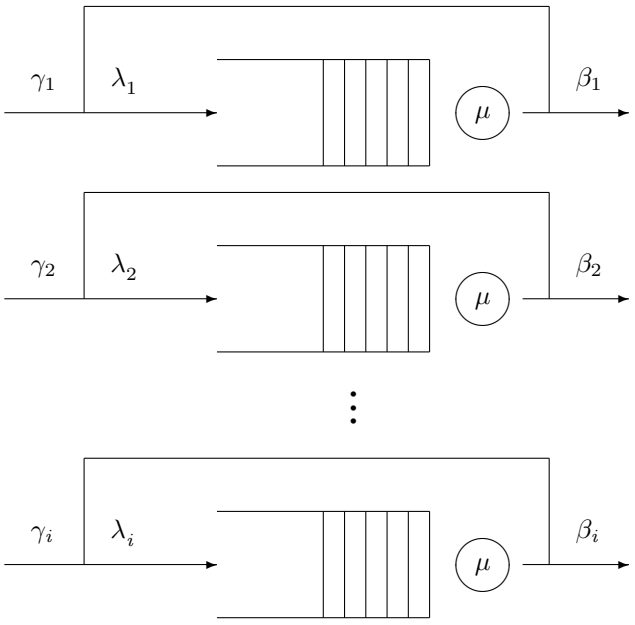


Figure 4.12: A system with feedback and several servers, each of which has its own queue.

permutation, in which feedback goes to a shared queue but servers have individual queues for new threads is unlikely to occur and is not shown.

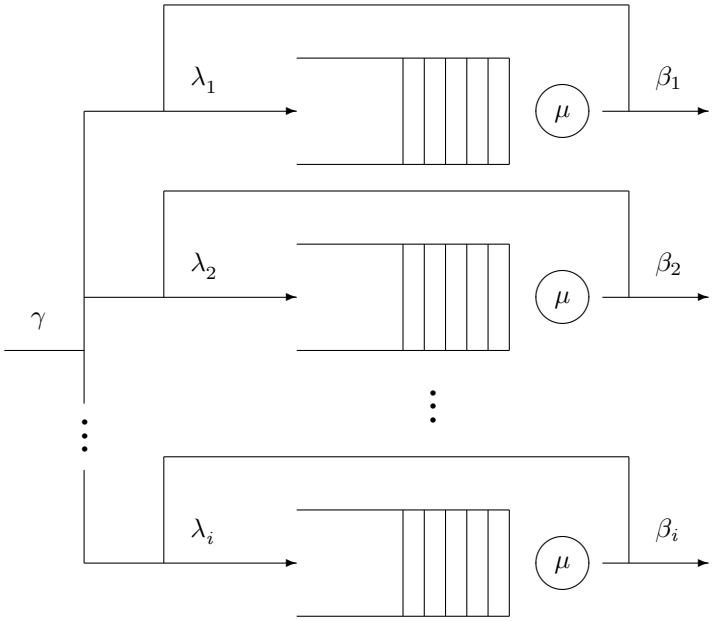


Figure 4.13: A system with several servers sharing a single queue, but with feedback on a per-server basis.

Stages of service

In this model, the execution of a thread is a multi-stage process, but the multi-stage process is achieved with only a single server; customers loop around to rejoin the queue until thread execution is completed.

4.3.2 Model analysis

Single-server model

For simplicity, we initially consider the single-server model. This is an example of what is known as an open Jackson network. A Jackson network is one in which all the exogenous inputs have Poisson distributions and the routing probabilities between nodes are known and state-independent. For each node i in the system, there is a mean arrival rate from the outside of γ_i, and a routing probability (once service is completed) to each of the other nodes j of r_{ij} [55]. For each node, the mean flow in must be equal to the mean flow out in a steady state. This condition may be expressed as a series of balance equations,

$$\lambda_i = \gamma_i + \sum_{j=1}^{k} \lambda_j \, r_{ji} \tag{4.2}$$

where k is the number of nodes in the system. This can be expressed more concisely in vector-matrix form as

$$\boldsymbol{\lambda} = \boldsymbol{\gamma} + \boldsymbol{\lambda}\,\boldsymbol{R} \tag{4.3}$$

where $\boldsymbol{\lambda}$ is a vector of the mean flow rates, $\boldsymbol{\gamma}$ is a vector of the exogenous arrival rates, and $\boldsymbol{R}$ is a matrix of the routing probabilities.

In the single server case, there is a very simple description for the system, since there is only one node. The balance equations require that

$$\lambda_1 = \gamma_1 + \lambda_1 r_{11} \tag{4.4}$$
$$\tag{4.5}$$

and thus that

$$\lambda (1 - r) = \gamma \tag{4.6}$$
$$\lambda = \frac{\gamma}{1 - r} \tag{4.7}$$
$$\tag{4.8}$$

with

$$r = \frac{\lambda - \beta}{\lambda} \tag{4.9}$$

It is now possible to work out the mean number in the system using the result

$$L_i = \frac{\rho_i}{1 - \rho_i} \tag{4.10}$$

where $\rho_i = \lambda_i / \mu_i$. This gives

$$L = \frac{\frac{\gamma}{\mu(1-r)}}{1 - \frac{\gamma}{\mu(1-r)}} \tag{4.11}$$
$$= \frac{\gamma}{\mu(1 - r) - \gamma} \tag{4.12}$$
$$\tag{4.13}$$

For an M/M/1 queue, Little's law states

$$\lambda W = L \tag{4.14}$$

where W is the mean waiting time in the system and L is the mean number in the system. This result is also applicable to Jackson networks, and so

$$W = \frac{\gamma}{\mu(1 - r) - \gamma} \frac{1 - r}{\gamma} \tag{4.15}$$
$$= \frac{1 - r}{\mu(1 - r) - \gamma} \tag{4.16}$$
$$\tag{4.17}$$

In the case when $r = 0$ (no feedback) this reduces to the form for a conventional M/M/1 queue,

$$W = \frac{1}{\mu - \lambda} \tag{4.18}$$
$$\tag{4.19}$$

More interesting than the wait time for a single instruction execution time is the sojourn time, that is, the time taken from when a thread enters the queue to when it completes execution and terminates. The sojourn time is analogous to the response time of the system, since providing a response will in general involve executing a number of instructions. The mean time between entrance into the system and final exit from the system is more complicated to calculate, because of the feedback. If a customer experiences a particularly long service time, it is likely that more customers will build up behind it

in the queue, and so when it rejoins the end of the queue it will have an extra long wait for its next stage of service, exacerbating the original delay [55].

In their survey of queueing network results, Disney and König [35] give the sojourn time distribution for an M/G/1 variant of simple instantaneous feedback network shown in figure 4.9. The probability of a sojourn time exceeding a time t is

$$\Pr[S_m \leq t] = \sum_{k=1}^{\inf} \pi \mathcal{Q}_i^{(k)} p q_{k-1} U \tag{4.20}$$

where $m = 1, 2 \ldots$ is the customer number, S_m, is the sojourn time of that customer, π is the stationary distribution for the embedded queue length process $\mathcal{N}^i$, $\mathcal{Q}_i^{(k)} k(t)$ is the matrix of k-step transition functions

$$\Pr[N(t_{mk}^i - 0), t_{mk}^i \geq t \mid N(t_{m1}^a - 0) = i, K = k], \tag{4.21}$$

$p = \Pr[\, Y_m = 1]$ and U is a vector of all whose elements are 1. Even in the single-server case this is not a particularly tractable solution, and is unlikely to allow much insight into the effects of garbage collection.

4.3.3 Transaction-level queueing

Things simplify considerably if feedback is omitted from the model. Instead of representing customers as threads requiring repeated processing by the server, the customers are chains of instruction requets, representing a single transaction or item of work. In this much simpler model, corresponding to the queue shown in figure 4.5, progress can be made in modelling garbage collection.

Garbage collection can be modelled in one of two ways. It can be treated as a high-priority customer, or it can be modelled as a periodic interruption. These two models are in fact quite closely related. White and Christie [93] provide an analysis of the effect of pre-emptive priorities in a queueing system. When a customer with a higher priority joins the queue, the customer currently in service is removed from service and the higher priority customer takes its place. They point out that when the number of classes is two, the premptive priority model can be interpreted as a model of a system with service interruptions (breakdowns), with interruptions possible while an item is in service. By extension, a head-of-queue priority system, in which items are not removed from service when a higher priority class joins the queue, is equivalent to a service interruption model where interruptions do not occur while an item is in service. White and Christie point out that for full equivalence, it is in general necessary to restrict the number of high priority customers to one or less — that is, no breakdowns should be building up while the server is already broken down.

Preemptive priority model

We consider the case where garbage collection is treated as a high priority customer. The arrivals at the processor are made up of two independent streams of traffic. The mutator stream has arrival rate λ_m and service time S_m. The garbage collection has arrival rate λ_g (the inverse of the collection interval) and service time S_g (the mean pause time, or the inverse of the service interval, μ_g). The garbage collection jobs have higher priority, going to the head of the queue and evicting application jobs from service. The application jobs resume service at the point they stopped, so this queueing discipline is called preemptive resume.

The assumption that the the two streams of traffic are independent is optimistic to the point of incorrectness. One effect this model omits is the negative feedback between garbage collector performance and the throughput. When the system is processing data rapidly, more data will be generated, and the garbage collection will be required to run more frequently, damping the performance of the system. In the edge case, if there is no application traffic, no garbage collection will be required and λ_g will be 0. Therefore the assumption that the two streams are independent is clearly deviating from the realities of the situation. However, modelling this kind of interaction would greatly increase the complexity of the calculations. This subject is revisited (without any further mathematical treatment) in section 6.3.

In general the mean response time for a customer of class p (where higher numbers indicate higher priority) is [85]

$$E[W_p] = \frac{1}{2(1-\sigma_{p-1})(1-\sigma_p)} \sum_{k=1}^{p} \lambda_k E[S_k^2] + \frac{E[S_p]}{1-\sigma_{p-1}} \tag{4.22}$$

where $\sigma_p = \sum_k^p \rho_p$.

This reduces to

$$E[W_g] = \frac{1}{2(1-\rho_g)} \left(\lambda_g E[S_g^2]\right) + E[S_g] \tag{4.23}$$

for the garbage collector threads and

$$E[W_m] = \frac{1}{2(1-\rho_g)(1-\rho_g-\rho_m)} \left(\lambda_g E[S_g^2] + \lambda_m E[S_m^2]\right) + \frac{E[S_m]}{1-\rho_g} \tag{4.24}$$

$$= \frac{1}{1-\rho_g} \left(\frac{\lambda_g E[S_g^2] + \lambda_m E[S_m^2]}{2(1-\rho_g-\rho_m)} + E[S_m] \right) \tag{4.25}$$

for the mutator threads.

These results only assume an M/G/1 queue, so the requirement that service times be exponentially distributed can be relaxed. In the two-priority case, the derivation using mean value analysis is relatively straightforward. As long as the system is conservative and work is not repeated (as it would be if service resumed at the beginning after an interruption), Little's law holds. For a wait time, W and occupancy, L, Little's law gives

$$\lambda_m E[W_m] = E[L_m] \tag{4.26}$$

$$\lambda_g E[W_g] = E[L_g] \tag{4.27}$$

Intuitively, the wait time for the high priority garbage collection jobs is given by

$$E[W_g] = E[T_{0g}] + E[T_g] + E[S_g] \tag{4.28}$$

where T_{0g} is the residual service time of the job in service, should it be a high priority job, and T_g is the service time of all high-priority jobs ahead of the current job, excluding the one in service. Since ρ_g is the probability of a garbage collection job being in service, the expected number of jobs is $L_g - \rho_g$. The expected time excluding the job in service is then $T_g = (L_g - \rho_g)/\mu_g$.

It would appear sensible to set T_g to zero, since a garbage collection should never have to wait for a garbage collection to complete before running, since if a garbage collection is running, no further collections are required. Unfortunately, limiting the capacity of the system in this way would mean arrivals were no longer Markovian and complicate the model. Moreover, some garbage collection systems in fact do *not* make the simplifying assumption that only one garbage collection should be trying to run at any one time; in particularly pathological cases logs show collections queueing up. Setting the arrival rate of the collections to be low relative to the service times ensures that garbage collections do not queue in almost all cases, which in fact corresponds quite neatly to physical reality.

The residual lifetime of a job with mean service time X is given by

$$E[R] = \frac{1}{2} \mu E[X^2] \tag{4.29}$$

The dependency on the variance is introduced because a job is more likely to arrive during while something with a long service time is being process rather than something with a short service time. In this case these times must be multiplied by the probability of finding a job of a given priority in service, ρ_p, giving

$$E[T_{0g}] = \frac{1}{2} \lambda_g E[S_g^2] \tag{4.30}$$

$$E[T_{0m}] = \frac{1}{2} \lambda_m E[S_m^2] \tag{4.31}$$

Using $\mu_g = 1/E[S_g]$ and $\rho_g = \lambda_g / \mu_g$,

$$\begin{aligned} E[W_g] &= E[T_{0g}] + \frac{E[L_g] - \rho_g}{\mu_g} + E[S_g] && (4.32)\\ &= E[T_{0g}] + \frac{\lambda_g}{\mu_g} E[W_g] - \lambda_g + E[S_g] && (4.33)\\ &= \frac{E[T_{0g}] + (1-\rho_g)E[S_g]}{1-\rho_g} && (4.34)\\ &= \frac{\lambda_g E[S_g^2]}{2(1-\rho_g)} + E[S_g] && (4.35) \end{aligned}$$

For the mutator threads, the expected wait time is

$$E[W_m] = E[T_{0g}] + E[T_{0m}] + E[T_g] + E[T_m] + E[T_p] + E[S_m] \quad (4.36)$$

where T_{0g} and T_g are as before, T_{0m} is the residual service time of the job in service, should it be a low priority job, T_m is the expected service time of all mutator jobs ahead of this one in the queue, excluding any in service, and T_p is the expected service time of all garbage collection jobs that arrive during time W_m and preempt the mutator job. If the mutator job is still in the queue, they will preempt by inserting themselves in front of the job in the queue, while if the job is in service, they will temporarily evict the mutator job from service.

Considering first the $E[T_{0g}] + E[T_{0m}] + E[T_g] + E[T_m]$ terms, these sum to the average waiting time in a non-preemptive queue, and can be calculated using the sum of the residual times and the wait times weighted by the server load, σ_k.

$$\begin{aligned} Q &= E[T_{0g}] + E[T_{0m}] + E[T_g] + E[T_m] && (4.37)\\ &= \sum_{i=1}^{k} E[R_i] + Q \sum_{i=1}^{k} \rho_i && (4.38)\\ &= \frac{\lambda_g E[S_g^2] + \lambda_m E[S_m^2]}{2(1-\rho_g-\rho_m)} && (4.39) \end{aligned}$$

T_p can be calculated using the expected service time of the garbage collection jobs and their expected arrival rate, and multiplying by $E[W_m]$.

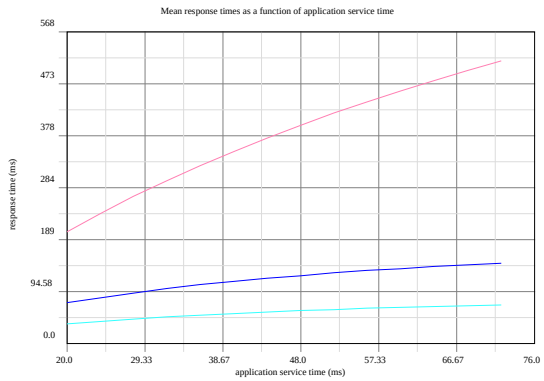
$$\begin{aligned} E[W_m] &= Q + \frac{E[W_m]\lambda_g}{\mu_g} + E[S_m]\\ &= \frac{\lambda_g E[S_g^2] + \lambda_m E[S_m^2]}{2(1-\rho_g-\rho_m)} + \rho_g E[W_m] + E[S_m]\\ &= \frac{1}{1-\rho_g}\left(\frac{\lambda_g E[S_g^2] + \lambda_m E[S_m^2]}{2(1-\rho_g-\rho_m)} + E[S_m]\right) \end{aligned}$$

Neglecting for the moment the second order terms in equation 4.25, the response time for the application threads is determined by their mean service time and the product of the frequency and duration of the garbage collections. Small garbage collection pause times are not sufficient if the frequency of collections is very large. In the limiting case where ρ_g approaches 1, the application response times will approach infinity.

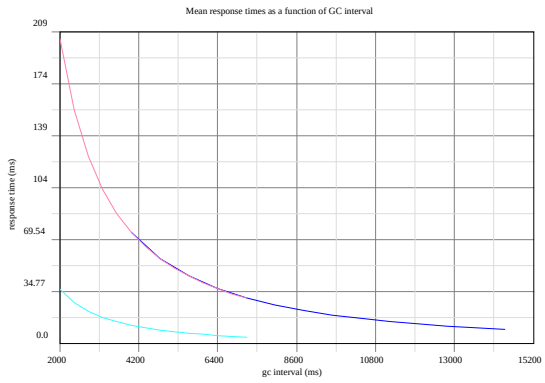
Figure 4.14 shows some example numerical results. The shape of the curves depends heavily on what values are chosen for the intervals and service times. While the application response times are sensitive to the pause time, they are also sensitive to the interval between pauses and the application service time.

The response times are always greater than the application service times. This is a fundamental property of queueing systems with Markovian arrivals. There is an asymmetry in how easy it is to grow and shrink the queue. The server cannot reduce the queue below length zero during quiet periods, but new arrivals can grow the queue no matter when they arrive. This means that new arrivals are much more likely to see a queue than an empty server, and it is necessary to make the server process new arrivals at a rate much greater than their arrival rate in order to avoid very large queues.

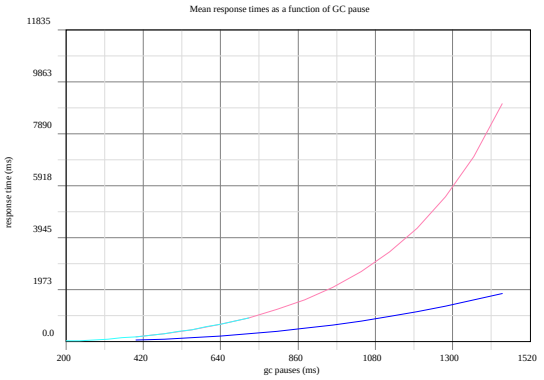
The SPECjbb benchmark deviates from this model because it processes work continuously for a predetermined period of time. The queue length is fixed at N, the number of warehouses, rather than being allowed to vary as work arrives. This can be reflected in the model by setting T_m to $NE[S_m]$.



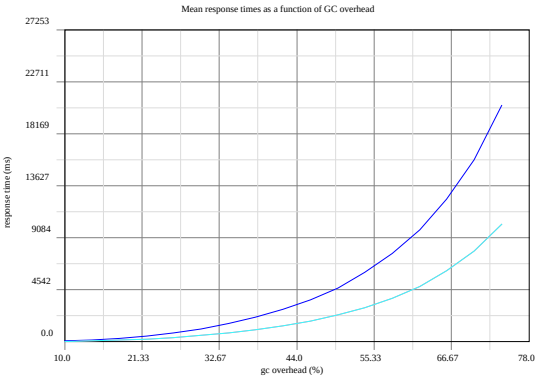
(a) Mean response time as a function of application service time.



(b) Mean response time as a function of garbage collection interval.



(c) Mean response time as a function of garbage collection pause.



(d) Mean response time as a function of garbage collection overhead (ρ_g).

Figure 4.14: Numerical results based on equation 4.25. The cyan line had the application service time second moment set to 10 ms^2, the gc pause time second moment set to 100, the mean pause time set to 400 ms, the mean service time set to 20 ms, the gc interval set to 4000 ms, and the thread interval set to 400 ms. The blue line had the interval adjusted up to 2000 ms, while the cyan line had the mean pause time reduced to 200 ms. The dependency on the second order moments is linear and not plotted.

Processor sharing model

Although the preemptive resume model is mathematically compact, it does not really capture the thread scheduling which the feedback model did. An alternative model is a processor sharing model with service interruptions. In a a processor-sharing queue each of k processes gets a fraction $1/k$ of the processor time. The wait time distributions between processor sharing and random order of service disciplines are equivalent [22].

The suitability of several different models is not coincidental. Van den Berg and Boxma showed that an M/M/1 queue with feedback (such as the one shown in figure 4.9) was equivalent, in the limit of high feedback probabilities and small service times, to an M/G/1 processor sharing queue [87].

An analysis of a processor-sharing system with service interruptions is given by Nuñez-Queija [76]. His solution is extremely complex and is not reproduced here.

In a garbage collected system, there are two factors which affect how quickly threads are served. The first, obvious, one is that garbage collections occur in which no application instructions will be executed and all customers must queue. These correspond to server breaks in a queueing model. Concurrent collection techniques can reduce the pauses, but at the cost of generally worsened service times. (Each application thread must assist with the garbage collection tasks every time it gets served, effectively paying a tax.)

The service time is an important second factor in determining the general level of satisfaction of the application threads. Service times can be explicitly affected by concurrent collection techniques, when every application thread pays a tax. They are also affected by much more subtle factors relating to the memory architecture of the system. These factors are discussed in chapters 5 and 6.

Chapter 5

An increase in pause times indicates things have got worse

Nature knows no pause in progress and development, and attaches her curse on all inaction.

Johann Wolfgang von Göthe

When assessing the performance impact of an algorithm change, or comparing different virtual machines, mean pause times are often used as the metric of GC quality. This can sometimes give results which are the exact opposite of correct. In the most trivial cause, reduced collector concurrency will give increased pause times, but also improved performance. Reductions in collection frequency can improve application performance at the cost of increased pause times. More subtly, often the worst-case properties are of greater importance than the average-case properties; optimising the average-case properties can lead to a degradation in the worst-case properties.

5.1 Defining 'things'

The performance of a system can be categorised in terms of two metrics: response times and throughput. Response time (or latency) is an indicator of the amount of time between requests being made and responses being received. Throughput refers to the amount of data which is processed in a given time.

Before deciding whether things have worsened or improved, it is important to decide what 'things' are in terms of these two metrics. For some systems, response times will be critical, and throughput less important. Real-time systems fall into this category. For other systems, response times are irrelevant but throughput is important. An overnight batch processing system is an example of the latter. In many cases response times are conflated with throughput and inappropriate system tuning decisions are made as a result.

For many applications, response times rather than throughput are the important performance characteristic. For some applications, such as ones with user interaction, there is almost no throughput requirement at all. For other applications, such as online trading and investment banking applications, there is a minimum throughput requirement as well. For example, a banking application may need to be able to respond to changes in share prices extremely quickly. While throughput isn't as important as response time, the application must also be capable of handling a reasonable application load.

The trade-off between throughput and predictability will be familiar to all who travel by train. If all trains were scheduled to travel at their maximum speed, the number of trains per day would be maximal and the train throughput would be very high. However, once a train was delayed, it would have no reserve speed left to try and make up the time and it would remain delayed until at least the next day. The platform scheduling involves a similar balance. If trains are intended to occupy platforms back-to-back, a delay to one train would cause serious problems. If the order of trains is

preserved, all trains will be held up until the delayed train arrives. If the delayed train is skipped and the later trains run as scheduled, then the delayed train won't be able to get a platform slot until another train is delayed and misses its slot. This wait for a platform could turn a minor delay into an extremely long delay.

While train passengers would undoubtedly like trains to travel as fast as possible and arrive in stations as frequently as possible, the reliability of the trains is also important. Passengers with connections or important appointments left standing on platforms for hours waiting for non-deterministically scheduled trains are unlikely to appreciate the high throughput statistics of the train network. On the other hand, they are also unlikely to appreciate trains which arrive precisely on schedule but travel at a walking pace, or stations which have only two scheduled services a day.

When defining 'things', it is also necessary to decide whether the average case, worst-case, or best-case metric is being considered. For most systems the average case is most important, but for some systems, particularly real-time systems where the consequences of a failure may be catastrophic, the worst-case is the most important.

Benchmarks inevitably include one or more performance metrics which can, with relatively little thought, be used to assess the effects of garbage collection changes. However, for many real applications, clarifying the desired performance properties is not easy, because there is no well-defined performance metric. Many real-world applications fall into this category.

If there is no throughput metric available the pause times are often the only measure available of the impact of GC changes. In these cases making any kind of inference about application performance is both difficult and dangerous.

The relationship between pause times and a response time metric is discussed in chapter 4. The conflation of responsiveness and throughput is particularly common when discussing real time systems. It is a popular misconception that, because of their high responsiveness, real time systems offer high performance. If the true performance criteria for a system is throughput, this is completely untrue. The aphorism "Real time is not real fast' [8] summarises the point neatly. In heavily overloaded systems, the poor throughput properties of real time systems can even lead to response time degradations, as discussed in chapter 4.

5.2 The relationship between individual pauses and the total pause

Changes to the collection frequency can significantly affect pause times. Figure 5.1 shows pause times for three runs of the SPECjbb benchmark. The mean pauses are summarised in table 5.1. The larger heaps are associated with larger mean pauses, although there are more time-consuming compactions (the spikes in the plot) with the smaller heap.

For mark-sweep collectors, pause times tend to be proportional to the size of the heap, rather than the amount of live data. Decreasing the heap size therefore reduces the pauses, and apparently makes 'things' better. However, plotting the pauses as a function of the collection number (figure 5.2), rather than time, suggests a different story. It's clear that the area under the pause plot for the larger heap is much smaller than the area under the plot of the smaller heaps. In fact, the run with the 500 MB heap spent more than half its time garbage collecting, while the run with the larger heap used only 8% of the time collecting.

The issue is that the collector must collect much more frequently with the smaller heap, and the increased frequency of collection undoes any benefit of the smaller pauses. The intervals between collections for the three runs are shown in figure 5.3.

The intuition that collecting so much more frequently cannot be beneficial is confirmed by comparing the reported throughput for the three runs. The thoughput is significantly higher with the larger heap. Even mean and maximum response times are similarly better with the larger heap. The SPECjbb performance metrics are plotted in figure 5.4.

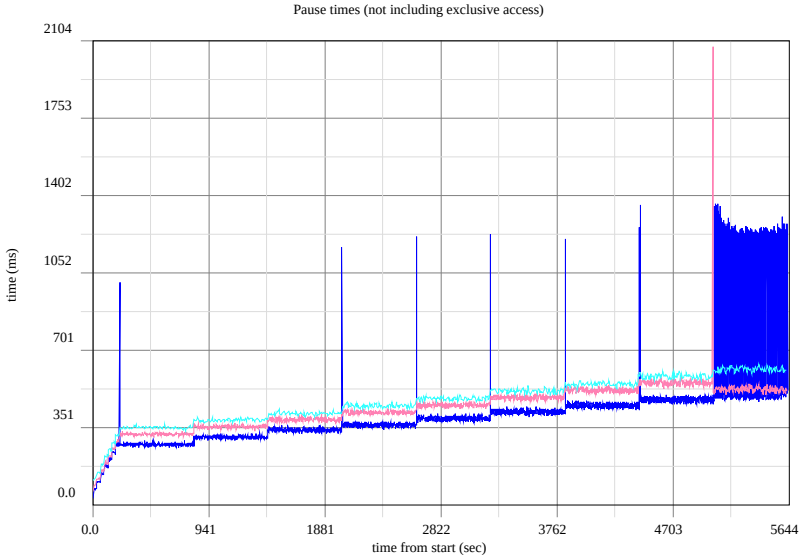


Figure 5.1: Pause times for three runs of the SPECjbb2005 benchmark. The heaps were fixed at 500 MB (blue line), 1000 MB (pink line), and 2000 MB (cyan line). As the heap increases, the mean pause increases.

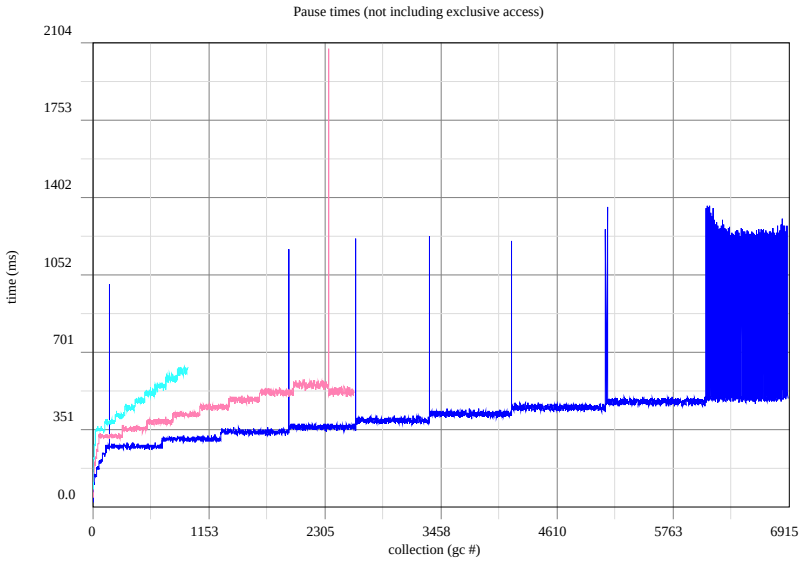


Figure 5.2: The same pause times as shown in figure 5.1, but plotted as a function of the collection number. The heap sizes were fixed at 500 MB (blue line), 1000 MB (pink line), and 2000 MB (cyan line).

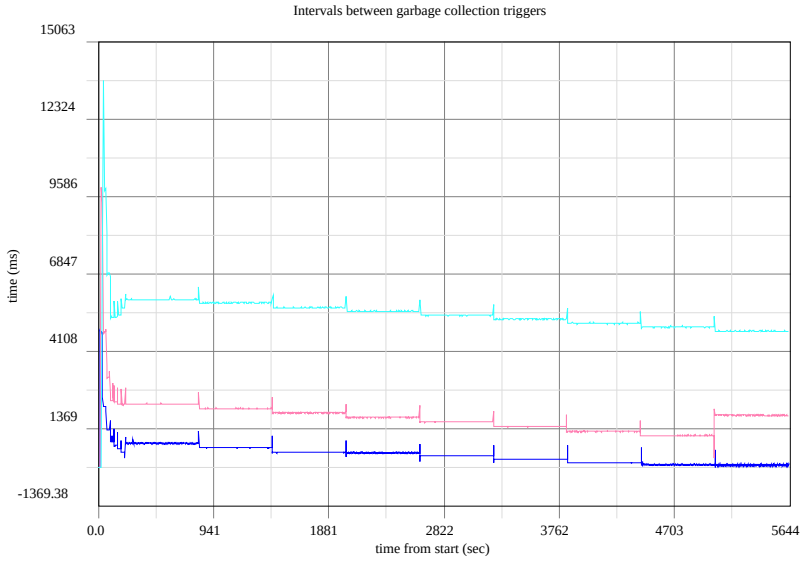
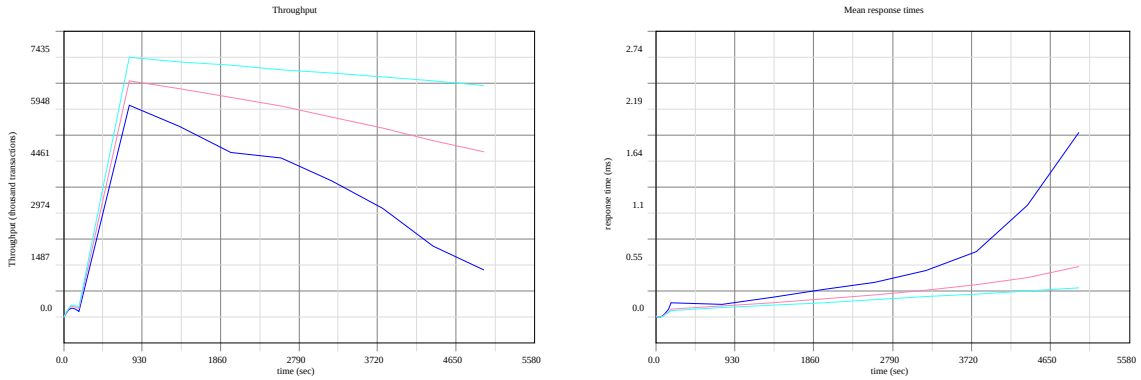


Figure 5.3: The same intervals between collections for the three runs shown in figure 5.1. The heap sizes were fixed at 500 MB (blue line), 1000 MB (pink line), and 2000 MB (cyan line).

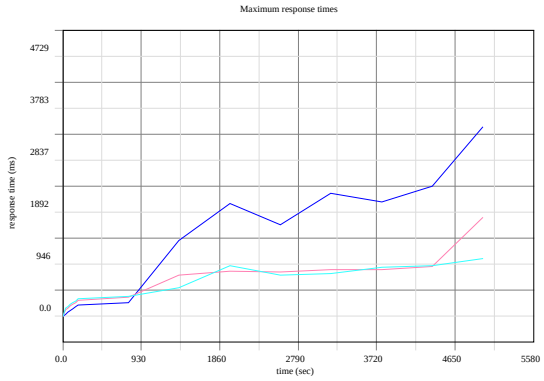
Variant	mean pause (ms)	mean interval (ms)	total pause (sec)	mean throughput (ktransactions)
500 MB heap	429	395	2967	2092
1000 MB heap	453	1707	1178	3387
2000 MB heap	481	5411	457	4036

Table 5.1: Several performance properties for SPECjbb runs in heaps of different sizes. While the mean pause is best in smaller heaps, the smaller intervals between collections mean performance is actually best with the larger heaps.



(a) Throughput

(b) Mean response times (per warehouse)

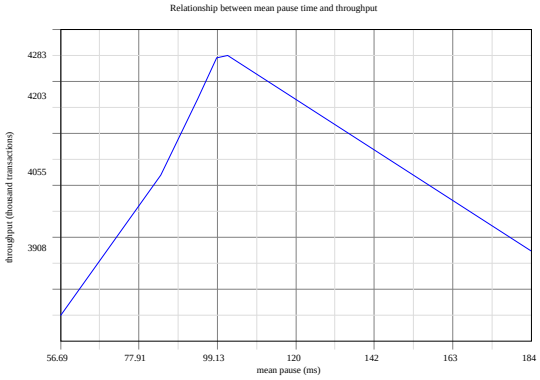


(c) Maximum response times (per warehouse)

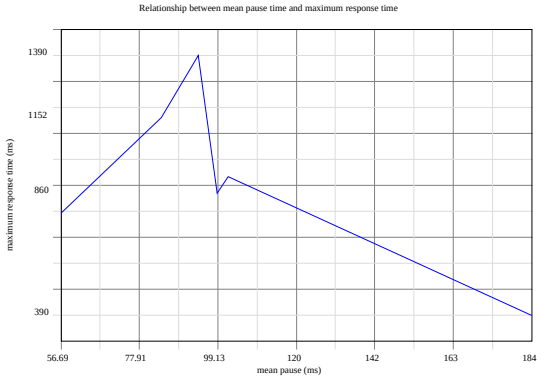
Figure 5.4: Throughput, mean response times, and maximum response times for SPECjbb2005 runs with three different heap sizes. The heap sizes were fixed at 500 MB (blue line), 1000 MB (pink line), and 2000 MB (cyan line). These results go along with those shown in figure 5.1.

5.2.1 Nursery sizing

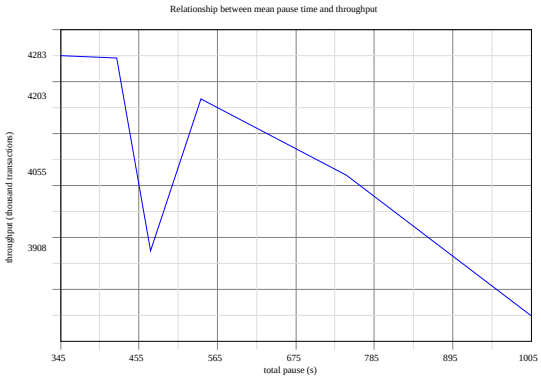
As with flat heaps, reducing the nursery size in a generational collector reduces the pause times. The relationship between the mean pause and the throughput is more complex than it is for flat heaps, largely because the relationship between the mean pause and the total pause is more complex. Figure 5.5 shows the relationship between the mean pause time, the throughput, the maximum response time, and the total pause time for a SPECjbb2005 run in which the nursery was varied from 100 MB to 700 MB, with the heap fixed at 1000 MB. The best throughput was achieved when the heap was 500 MB, which is also when the total time paused was smallest. For small nurseries, the throughput is inversely proportional to the mean pause, while for larger nurseries, the throughput is proportional to the mean pause.



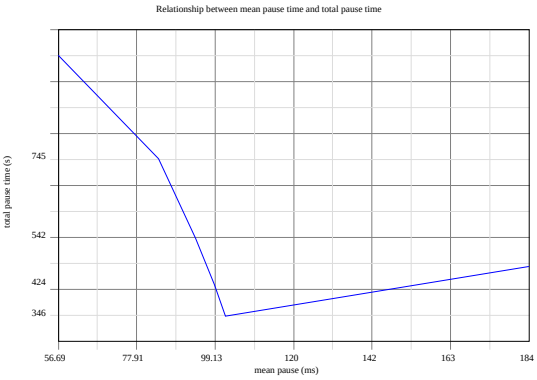
(a) Throughput as a function of mean pause.



(b) Maximum response time as a function of mean pause.



(c) Throughput as a function of total pause.



(d) Total pause time as a function of mean pause.

Figure 5.5: The relationship between the mean pause time, the throughput, the maximum response time, and the total pause time for a SPECjbb2005 run in which the nursery was varied from 100 MB to 700 MB, with the heap fixed at 1000 MB.

5.3 Worst case and average case

When tuning a garbage collection system, it is important to define the performance properties of interest and avoid optimising properties (such as mean response time) which are unrelated to the

properties which are genuinely of interest, such as throughput or response time. It also important to be clear on whether the mean or worst-case properties are of interest.

Real time systems place the priority on determinism over speed and have a tendency to over-provision. They emphasise the worst case, rather than the average case as most systems do. This emphasis on the worst-case can require some onerous tuning decisions, and so a distinction is sometimes made between hard real-time and soft-real time. Hard real time has a strict worst-case criteria, while soft real-time allows some worst-case possibilities to be ignored and some average-case properties to be taken into account. Soft real-time is suitable for systems with response time criteria for which the consequences of failure are not unrecoverable.

Metronome is a novel mostly non-copying collector [10]. It does some copying, in order to avoid the potentially unbounded fragmentation experienced by non-copying collectors. However, it significantly overlaps the from- and to-spaces to avoid the space overhead of normal semi-space copying collectors. The intersecting from- and to-spaces require metronome to uses a read barrier and pointer-forwarding, which have an instruction cost for the mutator.

So irrelevant is throughput in the real time system, Bacon does not even report throughput figures for his collector, much less provide comparisons with traditional collectors running on the same hardware. This is a notable omission since the benchmarks from the SPECjvm suite he runs do have well-defined throughput metrics. He does quote a 4% average cost for his read barrier, but this is unlikely to represent the entire performance difference between the real time collector and a standard collector.

5.3.1 Work-based and time-based collection

Bacon makes a distinction between two classes of real-time collection, work-based and time-based. Both interleave mutator activity with non-concurrent collector activity, and rely on fine interleaving to achieve real-time targets.

In a work-based collector, the rate of collection is related to the rate of allocation. After some amount of allocation, Q_W, the collector runs and collects C_W of data. It must be the case that $Q_W < C_W$ or the space used by the application will grow without bound. The size of Q_W and C_W determine how finely the collections are interleaved with the mutator activity.

In a time-based collector, the collector and mutator are interleaved on a rigidly-scheduled basis. The mutator is allowed to run for Q_T milliseconds before handing control over to the collector, which also runs for C_T seconds. One consequence of this strict scheduling is that periods of unusually high allocation activity will cause the heap to grow.

Mean pause times for the work-based and time-based collectors are shown in table 5.2. Comparing the two policies, the time-based collector gives average pause times which are dramatically worse than those of the work-based collector. From this evidence, it is tempting to conclude that 'things' are dramatically worse with the time-based collector, and that the work-based collector is the better policy.

Policy	Mean pause time (ms)
Work-based	4.4
Time-based	11.1

Table 5.2: Mean pause times for Bacon's time-based and work-based real-time collectors. The results are the mean results from five of the seven SPECjvm benchmarks, javac, jess, jack, mtrt, and db. Data quoted from Bacon et al. [10].

The accuracy of this conclusion depends on what *things* are. If the system property of interest is the average pause time, the work-based collector is indeed the better collector. If, as is more likely in a real-time system, the property of interest is the worst-case pause time, conclusions must not be drawn from the mean data. Maximum pause times are shown in table 5.3. There is significant variation

between the mean and maximum pauses for the work-based collector, and almost no variation for the time-based collector.

Another property of interest in real time systems is the minimum mutator utilisation, which was discussed in chapter 4. This is another worst-case property. If the the minimum utilisation is considered, the work-based collector is a dreadful performer compared to the time-based collector. During periods of heavy allocation, control is almost entirely passed over to the The MMU over the target interval was so bad that a larger interval was considered as well, but even over this interval, the MMU cannot match that of the time-based collector over the shortter interval. collector, ruining the real-time guarantees of the system. The MMU figures are shown in table 5.3.

Policy	Mean pause time (ms)	Maximum pause time (ms)	Minimum utilisation (u_w(22.2ms))	(u_w(50ms))
Work-based	4.4	23.6	0.0006	0.1636
Time-based	11.1	12.36	0.443	/

Table 5.3: Mean and maximum pause times as well as minimum utilisation for the time-based and work-based collectors. The minimum utilisation on the 22 ms interval for the work-based collector is so poor that data for a 50 ms interval is included as well. The results are the mean results from five of the SPECjvm benchmarks, javac, jess, jack, mtrt, and db. Data quoted from Bacon et al. [10].

In this case the increased average pause of the time-based collector is associated with reduced worst-case pauses and a favourable change in the system properties. A naive analysis which did not fully clarify whether the worse-case or average-case pauses were of interest would have misinterpreted the results.

In these examples, small mean pauses have indicated neither that the total overhead of the garbage collection was low, or that system performance was particularly optimal. There are many situations when the mean pause times do turn out to be correlated to the total pause times. In these cases low mean pause times do genuinely indicate that the amount of time spent doing garbage collection was low. Even there, however, there is often very little relationship between the pause times and performance. This is the subject of chapter 6.

Chapter 6

A large total pause time means the policy is worse than one with a smaller total pause time

I would advise you to keep your overhead down; avoid a major drug habit; play every day; and take it in front of other people.

James Taylor

Even leaving aside differences in the amount of data being processed by an application, the total proportion of time an application spends paused does not necessarily give any clue as to the throughput. The obvious example is concurrent collection modes, where shorter pauses are achieved by delegating much of the garbage collection work to the mutator threads. The garbage collection pauses are shorter, but only because the garbage collection work is being done (with reduced efficiency) while the application continues to run. More surprisingly, there is often very little correlation between the absolute — reported and unreported — amount of work being done by the collector and the application performance. The impact of the collector on mutator performance is often much more significant than simply the time consumed by the collector itself. Finally, when the workload of an application is not fixed, a felicitous choice of garbage collector can result in more data being processed, and consequently more garbage collection and an apparently higher garbage collection overhead.

6.1 Perceived overhead and actual overhead

The inadvisability of inferring that the small pauses in concurrent collection policies lead to good performance has already been discussed in chapter 4 and chapter 5. Not only are the small pauses no guarantee of good latency, the total time reported paused in the application is no indicator that the mutator has received its fair share of the CPU. A distinction must be drawn beween the reported total pause time and the actual overhead of garbage collection.

6.1.1 Hiding work

The first problem in trying to draw conclusions about application effectiveness from garbage collection pause times is that reported pause times often do not reflect the true cost of the garbage collection. Garbage collection pause times can be reduced by putting more of the collection burden on the mutator, effectively hiding the collection work in with the real application work. This is the mechanism used in concurrent collectors. The possibility of hidden work makes it very difficult to interpret pause time data accurately.

When an allocation request is made of a concurrent collector, it may satisfy the request, it may trigger a collection during which no more allocation requests will be satisfied, or it may satisfy the

request but also do some incremental collection work. This work is not included in the *tock* counts of the verbose gc monitor.

$$\begin{aligned} ConcurrentCollector = {} & allocateobject?o \rightarrow ConcurrentCollector \\ & \sqcap \\ & allocateobject?o \rightarrow begincollection \rightarrow CollectingConcurrentCollector \\ & \sqcap \\ & allocateobject?o \rightarrow freeobject!p \rightarrow ConcurrentCollector \end{aligned}$$

The experiments shown in figures 6.1 and 6.3 allowed the overhead of the concurrent collection to be adjusted hidden in a controlled way. More of the work was shifted into the hidden concurrent stage of the collection, leaving less for the explicit paused stage.

Many concurrent collectors use a card cleaning and dirtying mechanism [13]. The collector begins tracing with the roots and traces concurrently with the application activity. Since the tracing is concurrent, the mutator threads will be changing the object reference graph and invalidating the results of the tracing. To compensate for this, a write barrier is used to trap mutator writes and record that affected objects need to be retraced. Recording this information on a per-object basis would add an unacceptable space overhead, and so more granular units called cards are used. When an object is modified, the card containing that card is dirtied. As long as mutator activity continues, cards will be dirtied, and so the collector can never hope to clean all cards concurrently. Most collectors make one pass across the cards and then halt all threads and clean the dirtied cards. Once final card cleaning has been performed a concurrent sweep is done to construct a new free-list.

However, the number of passes need not be fixed at one. Figure 6.1 shows the impact on pause times of changing the number of card cleaning passes. If zero passes are made, the collector has all of the overhead of the concurrent algorithm with little of the benefit in terms of pause times. All cards must be cleaned in stop-the-world mode. When two passes are made, the number of cards which must be cleaned in stop-the-world mode is reduced. The disadvantage of the extra pass is that frequently-dirtied cards will be cleaned three times.

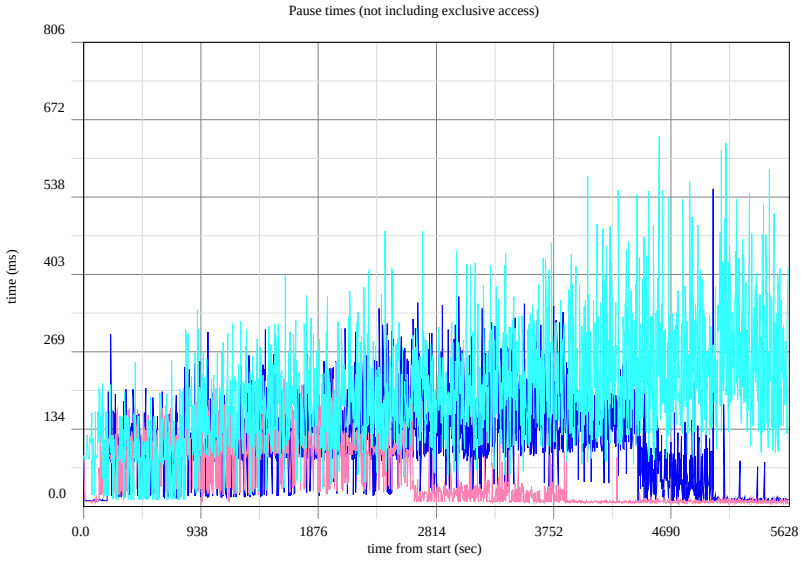
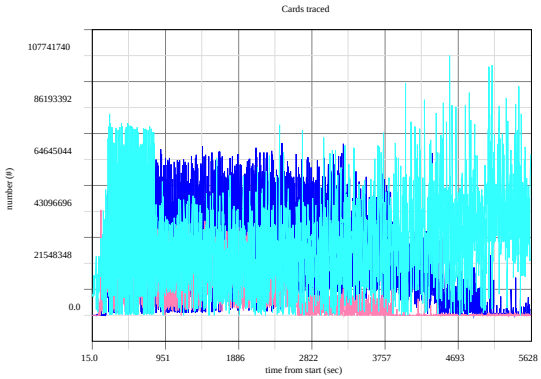


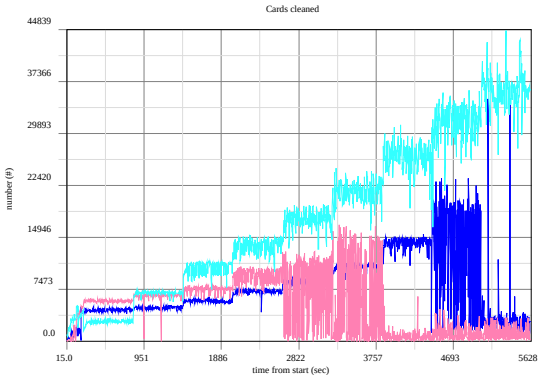
Figure 6.1: Reported pause times for three configurations of a concurrent collector, running SPECjbb2005. The cyan line did zero card cleaning passes per collection, the blue line did one, and the pink line did two. Pauses are significantly smaller when the number of passes is higher.

The pause times are quite closely related to the number of cards traced (figure 6.2(a)). The number of cards cleaned is more related to the number of warehouses (and threads). The greater the number

of warehouses, the more areas of memory will be touched by the mutator threads and the more cards will be dirtied. At some point after the number of warehouses exceeds the number of threads this effect is offset when cards are cleaned concurrently, but not when the number of card cleaning passes is zero. The number of cards cleaned is shown in figure 6.2(b).



(a) Cards traced non-concurrently. The mean when no passes were made was 30.3 million, when one pass was made was 25.0 million, and when two passes were made only 9.5 million cards were traced.



(b) Cards cleaned non-concurrently. The mean when no passes were made was 18.8 thousand, when one pass was made was 8.0 thousand, and when two passes were made 5.0 cards were cleaned. Each card cleaned involves tracing any cards referenced from that card, which is why the number of cards traced is so much greater than the number cleaned.

Figure 6.2: The number of cards traced and cleaned for three card cleaning policies. The number of cards traced is related to the pause times, while the number of cards cleaned is more closely related to the number of warehouses, with some deviations when the number of warehouses is large. The cyan line did zero card cleaning passes per collection, the blue line did one, and the pink line did two.

The description of the experiment has made it clear that work is simply being shifted from the concurrent phase to the non-concurrent phase; the lowered pause times do not represent miraculously lowered levels of garbage collection overhead. This is corroborated by the throughput results for the three configurations, shown in figure 6.3. Changing the number of card cleaning passes has remarkably little effect on the throughput. The mean throughputs are within 1% of each other, which is well within the statistical uncertainty. The pause and throughput results are summarised in table 6.1.

Card cleaning passes	Mean pause (ms)	Total pause (%)	Mean throughput (ktransactions)
0	221	6.6	2533
1	71	3.8	2556
2	9	1.6	2517

Table 6.1: Mean pause, total pause, and SPECjbb2005 throughput results for a Java virtual machine which has been instrumented to alter the number of card cleaning passes in its concurrent garbage collection algorithm. The idea and instrumented virtual machine are courtesy of Andy Wharmby [92].

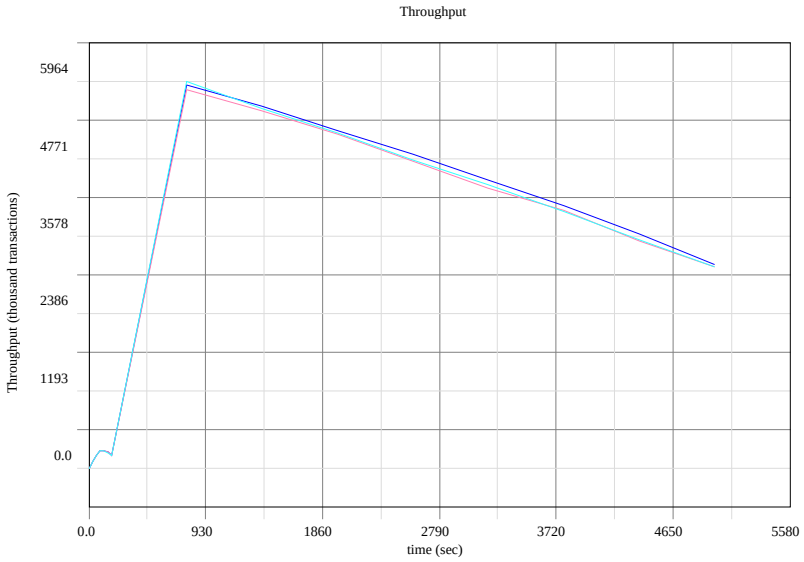


Figure 6.3: SPECjbb2005 throughput for three configurations of a concurrent collector. The cyan line did zero card cleaning passes per collection, the blue line did one, and the pink line did two. The mean throughputs are within 1% of one another.

6.1.2 Actual overhead

Table 6.2 shows the reported and actual overheads for a SPECjbb2005 run with concurrent and non-concurrent garbage collection policies. The experimental setup is the same as that of figure 4.2.

Policy	Reported overhead (%)	Actual overhead (%)
Concurrent	0.4	29.7
Non-concurrent	22.1	18.4

Table 6.2: Actual and reported total overheads and throughout for SPECjbb runs with concurrent and non-concurrent garbage collection policies. The reported overhead is based on the reported pause times, and the actual overhead is based on a profile collected with the TPROF profiling tool [60]. Only time in the gc library is reported, and so system calls are omitted, which may cause the overhead to be underrepresented.

The reported pause times are plotted in figure 6.4. Both the concurrent and non-concurrent collectors spent most of their time scanning and marking objects. For the non-concurrent collector, this accounted for about 14.5% of the total run time. The concurrent collector required 20% of the time to mark objects. The difference is probably because the concurrent collector is forced to remark objects in dirtied cards. The management of card dirtying and the write barrier — mapping heap addresses to card addresses and working out which card is associated with any particular object – contributed about 1% of the total cost of the concurrent collector. Both collectors spent 0.8% of the time pushing work onto and popping work off a work stack. This is part of the cost of parallelising the collection work across multiple threads.

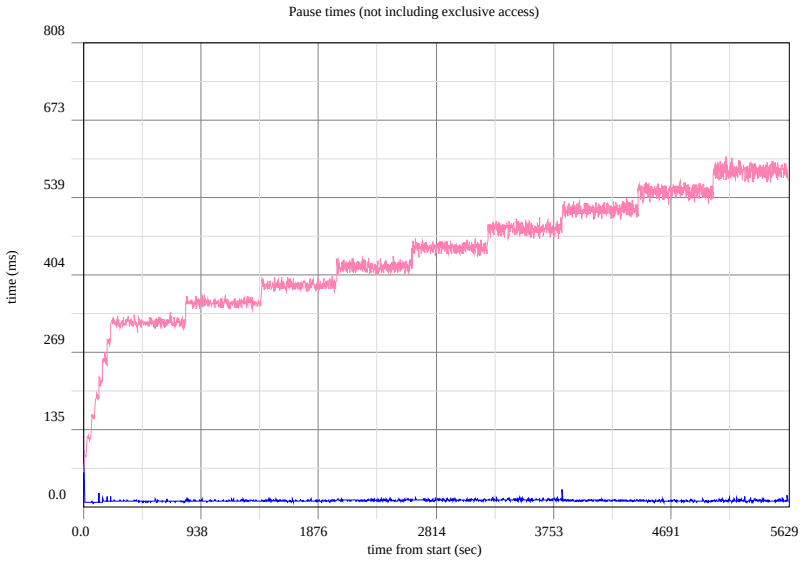


Figure 6.4: The reported pause times from a concurrent (blue line) and non-concurrent (pink line) SPECjbb2005 run, as summarised in table 6.2.

6.1.3 Overhead without concurrency

With some heap layouts, even stop-the-world collections have a hidden overhead. Generational collectors work by dividing the heap into multiple areas, and collecting only some areas at a time. While any number of generations is possible, most of the benefit is achieved with just one generation, which is also the simplest case. New objects are allocated in the nursery, which is usually a semispace collector. The nursery differs from a plain semispace collector in one respect, which is that when an object is deemed old enough, it is tenured. Tenuring old objects allows the collection in the nursery to focus on young objects which are likely to die, rather than wastefully copying old objects likely to live.

The added internal choice between copying and tenuring may be expressed using double renaming:

$$NurseryHeap(n, N) = SemispaceHeap(n, N)[moveobject \leftarrow moveobject, moveobject \leftarrow tenureobject]$$

The nursery and tenured heap synchronise on the tenure event.

$$GenerationalHeap = NurseryHeap \; [|\; tenureobject \;|] \; TenuredHeap$$

The tenured heap does not allow allocation, but tenuring is effectively the same operation.

$$TenuredHeap = FreeListHeap[tenureobject \leftarrow allocateobject]$$

If an attempt to tenure an object is made and the tenured heap is full, a global collection is performed.

The generational heap is refined by the general heap.

$$Heap \sqsubseteq_T GenerationalHeap$$

The awkward part of the generational collection is working out which objects in the nursery are live, since they can be referenced both from the application roots and from objects in the tenured area which won't be inspected by the collector. Inadertently collecting objects in the nursery which were referenced by objects in the tenured area would cause dangling pointer errors.

It is a fundamental assumption of the algorithm that references from old objects to young objects are rare. Those references that do exist are collected in what is known as a remembered set and used

as an additional root when collecting the nursery. In order to make sure that every reference to a young object from an old object is recorded, all writes to old objects are trapped with a write barrier. The write barrier adds an overhead to data changes which affect old objects.

When the heap is very large, the tenured area will not be collected often, and will include many references to young objects from old objects which are no longer live. This means young objects will be preserved unnecessarily. Nursery collections will be more frequent, and some young objects will be unnecessarily promoted as a result. Once in the tenured area, young objects no longer benefit from the favourable cache properties of generational collectors, and access to them is hampered by the write barrier.

The write barrier may be expressed as a synchronisation between the application and the heap on the *writeobject* event.

$$GenerationalMachine = \left(\|\| \; n : 1..N \bullet Processor\right) \; [|\, \alpha(Heap) \,|] \; GenerationalHeap$$

$$\begin{aligned} GenerationalGCSystem = \; & GenerationalMachine \\ & [|\, ApplicationActions \,|] \; (Application \; [|\, \{|freeobject|\} \,|] \; Stop) \\ & [|\, \{|freeobject, allocateobject, writeobject|\} \,|] \\ & GenerationalGarbageCollector \end{aligned}$$

This synchronisation naturally has the potential to slow the system down. (Concurrent collectors also require either a write barrier or a read barrier, which is a synchronisation with the *readobject* event. The write barrier is not discussed in the model of the concurrent collector given above but would be if the concurrent heap were modelled.)

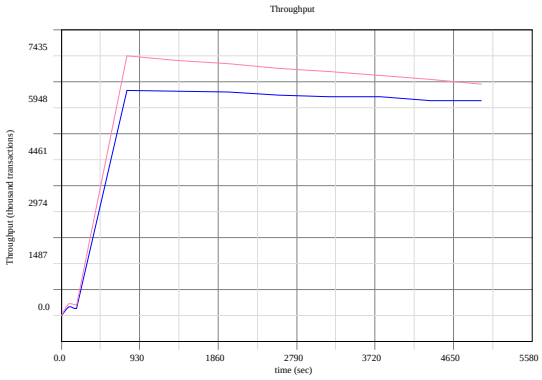
Policy	Mean pause (ms)	Total pause (min)	Mean throughput (ktransactions)
Mark-sweep	481	19.3	4036
Generational concurrent	42	7.6	3611

Table 6.3: Pause times and throughput results for a SPECjbb2005 benchmark with a mark-sweep and generational garbage collection policy. The heap was fixed at 2000 MB.

The throughput is 12% higher with the mark-sweep collector. The results are summarised in table 6.3.

While some garbage collection policies introduce hidden overheads, others introduce hidden speed-ups. The impact of the collector policy is much greater than just the work done, reported or unreported.

(a) Pause times for a generational concurrent collector and a mark-sweep collector. Pause times are significantly lower with the generational collector because collections of the small nursery can be performed very quickly.



(b) Throughput for a generational concurrent collector and a mark-sweep collector. The throughput is better with the mark-sweep collector.

Figure 6.5: Pause times and throughput for a generational collector (blue line) and a mark-sweep collector (pink line). While pause times are significantly lower with the generational collector, the throughput is 11% better with the mark-sweep collector. The heap size was fixed at 2000 MB.

6.2 The importance of mutator performance

In chapter 4, we discussed how small pause times were no guarantee of satisfactory performance if these pauses were spaced so closely together that the total time spent paused was quite large. When phrased in those terms, this is not a particularly surprising property. In this chapter we discuss cases when this property, despite not being particularly surprising, does not hold.

6.2.1 Object rearrangement

Collectors which arrange objects in the heap for optimum locality can significantly enhance the performance of the mutator. Often the rearrangement of the objects has a non-trivial overhead, either because the calculations involved in working out the optimum object positioning are reasonably intensive, or because the re-ordering occurs frequently re-ordered in order to ensure the layout is well-suited to the current pattern of application access. The extra overhead of the object rearrangement means there is often little correlation between total pause time and application performance.

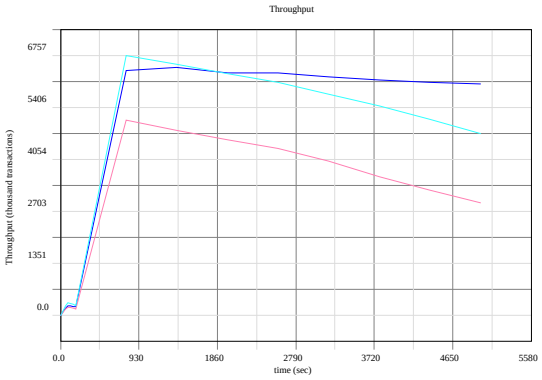
For memory intensive applications, memory performance has significant performance impact. Many CPU cycles are wasted in stalls waiting for memory access. Ali-Reza et al. measured that 45% of CPU cycles in the SPECjbb benchmark were spent waiting for memory requests.

For many workloads and system configurations, generational collectors perform extremely well. As discussed in chapter 3, allocation of new objects is much faster than it is with a free-list based allocator. There is no need to search for a gap large enough to accomodate new objects because objects are allocated in a completely empty new space. Because new objects are allocated contiguously, the cache properties tend to be optimal and references between these objects can be followed quickly. Finally, the minor collections of the nursery need only collect a small area, and the collection time is proportional to the amount of live data, rather than the area collected, so if the collections are timed well, pause times can be very small indeed.

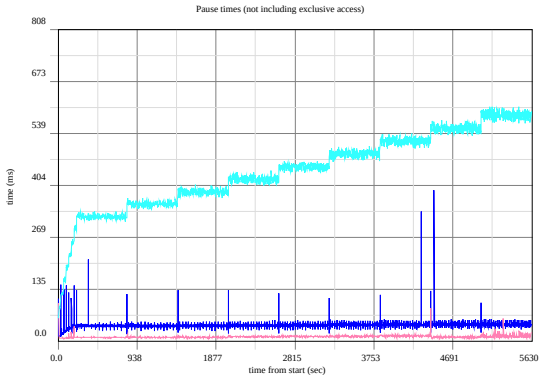
Generational copying collectors are a good example of one such moving collector. With some workloads and systems, an infeasibly large proportion of the total application time is spent paused for garbage collection, and yet the throughput exceeds that of the other modes. Generational collectors do not in general do any advanced application profiling, but nursery collections tend to be very frequent

and so they keep most of the working set of the application in a relatively small contiguous area of the heap.

Figure 6.6(a) shows the SPECjbb2005 throughput for three different collection policies. The inflection points in the mark-sweep and concurrent throughput occur when the collector compacted the heap and thus improved performance. The generational collector has the highest mean score, and the concurrent collector has the lowest. SPECjbb results are usually reported as a score, which includes a minimum response time criteria and a weighting of the throughput against the number of warehouses. Throughput is used in this discussion since it is a more transparent performance measure. Response time is discussed in chapter 4.



(a) SPECjbb2005 scores for three different garbage collection policies in a slightly constrained heap. The generational collector performs the best, with the mark-sweep collector 5% behind, and the concurrent policy lagging 36% behind.



(b) Garbage collection pauses for three different garbage collection policies. The optthruput collector has the highest mean pause, and the concurrent collector has the lowest.

Figure 6.6: Pause times and throughput for three different garbage collections. The blue line represents a generational policy, the pink line a concurrent policy, and the cyan line a mark-sweep policy. The heap was fixed at 1000 MB, giving a mean occupancy of around 50%.

The pause times, overhead, and throughput are summarised in table 6.4.

Policy	Mean throughput (ktransactions)	Mean pause (ms)	Number of collections	Total pause (%)
Concurrent	2268	13	1871	10.43
Mark-sweep	3290	463	2707	20.9
Generational	3573	42	26896	22.6

Table 6.4: The relationship between garbage collection policy, mean SPECjbb2005 score, and the total garbage collection pause. The heap size was fixed at 1000 MB. The experimental setup is identical to that of table 6.3 except that the heap is smaller.

Note that the variation in heap sizes produced results the opposite of those presented in table 6.3. Other combinations of heap size and runtime options can produce results in which the mark-sweep policy has the advantage over the generational policy, or for which the total time is greater for the mark-sweep mode than for the generational policy.

When the heap is large, many of the advantages of the generational collector are outweighed by disadvantages.

These factors mean that mark-sweep collectors perform substantially better in larger heaps, while the benefit is much less pronounced for generational collectors.

Changing the order in which objects are copied can give significant mutator benefits because of improved cache behaviour [82]. However the garbage collector tends to benefit from the improved locality as well as the application, and so these object reordering algorithms tend to benefit both mutator performance and garbage collection pause times.

6.2.2 Compaction

Performance increases through increased garbage collection overhead need not be achieved by sophisticated copying algorithms. Even indiscrimininate compaction can give performance benefits, despite disastrous effects on the overhead. Allocation is faster in a compacted heap because the free list is less fragmented. Object access is also faster because compacted objects are more likely to lie in the same cache line. Figure 6.7 shows the reported free heap (after collection) from two SPECjbb2005 runs. One run used a standard mark-sweep collector, while in the other the same collector was forced to compact on every collection. Normally the collector only compacts when it judges the heap to be particularly fragmented. This forced compaction eliminated enough fragmentation that the reported free heap was 60% higher in the run with the compactions.

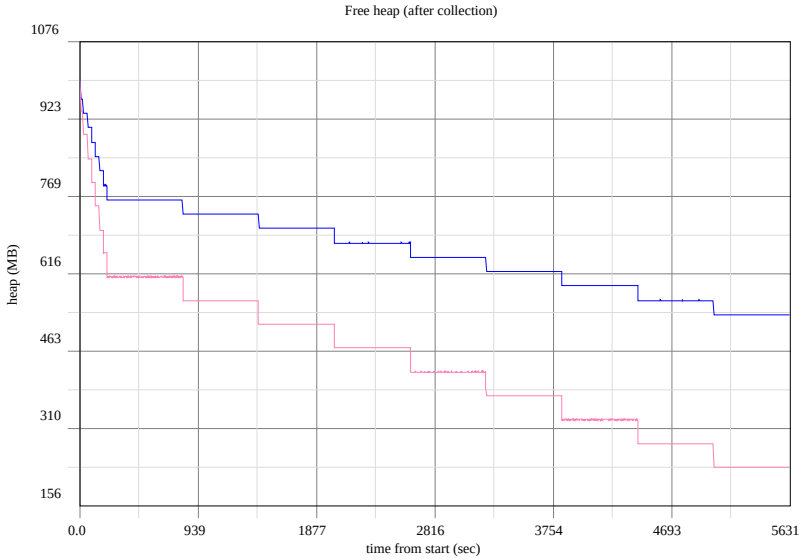


Figure 6.7: Reported free heaps for two SPECjbb2005 runs. One run used a standard mark-sweep collector, while in the other the same collector was forced to compact on every collection. The heap size was fixed at 1000 MB. Without forced compactions, the mean occupancy was 59%, while with the compactions it was 34%.

This extra heap does not come without a cost in terms of pause times, shown in figure 6.8. The mean pauses with the extra compactions are two times the pauses without the compaction. This is offset by the fact that more allocation was possible in the compacted heap before collection was required, so that the 102% increase in mean pause time only translates to a 33% increase in total pause time. The pause time overhead was 22.4% for the default collector, and 29.7% for the collector with the forced compactions. Although the mean pauses are much greater with the forced compaction, the mark and sweep phases were quicker. The mean mark time was 373 ms with the forced compaction, an 18% improved on the 441 ms required without compactions. Similarly, the sweep took an average of 19 ms instead of 26 ms.

The improved cache behaviour which allows mark and sweep times to be reduced by frequent compaction also results in a net throughput gain, despite the fact that less time is available to the application. Figure 6.9 shows the throughput results for the two runs. The throughputs are relatively comparable at the beginning of the experiment but as time passes and the uncompacted heap grows

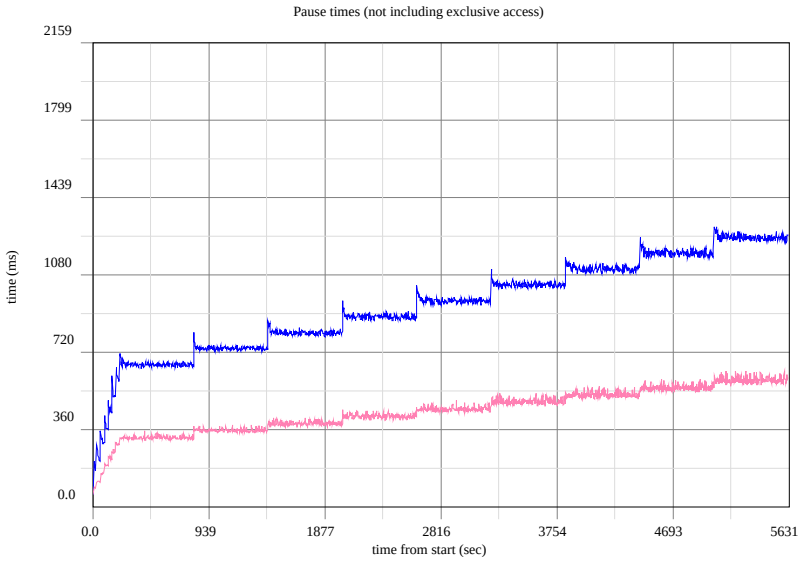


Figure 6.8: Pause times for the two SPECjbb2005 runs shown in figure 6.7. In one run, the collector was forced to compact every collection, resulting in significantly longer pause times. With forced compaction, there were fewer collections, so the total pause time is only 33% greater.

more fragmented, the application running with the compacted heap gains a clear advantage. Over the whole ninety minute run, the mean difference is 5% in favour of the forced compaction. By the final warehouse, the forced compaction is delivering an 18% benefit to the throughput.

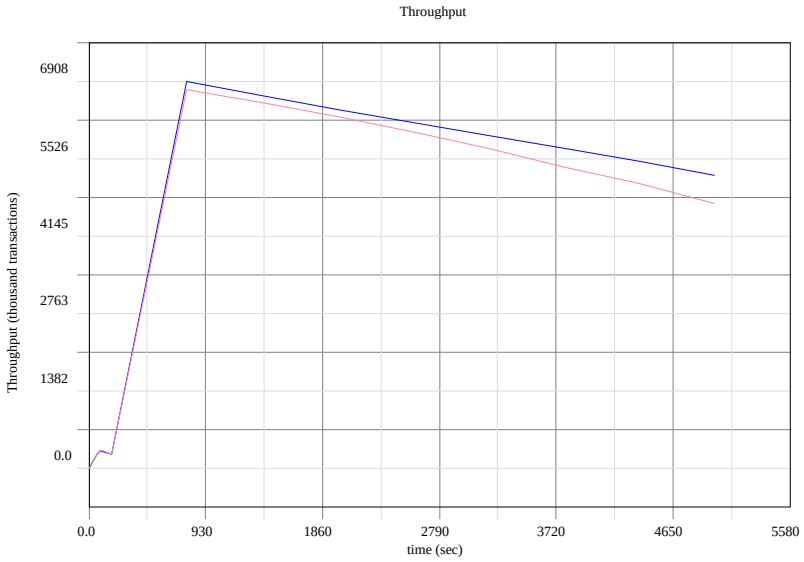


Figure 6.9: Reported throughput for the two SPECjbb2005 runs shown in figure 6.7. Forced compaction provides a mean throughput benefit of 5%.

6.2.3 Malloc/Free overhead and performance

The examples above have all used the SPECjbb benchmark, the same IBM Java virtual machine, and the same hardware. Since different applications have very different object allocation and access patterns, and since these patterns can interact with garbage collectors in unexpected ways, the SPECjbb illustrations cannot be taken to be representative of all applications. However, the non-correlation between memory management overhead and application performance is not restricted to Java, or even to languages which allow moving garbage collectors.

Table 6.5 shows the relative memory management overheads and performance of the Bohem-Demers-Weiser garbage collector and Ultrix C memory management library, as reported by Detlefs et al [31]. The BDW collector invariably spent more of the run instructions on memory management than the Ultrix collector. Intuition would dictate that when the BDW collector has particularly high overheads, the application time should be particularly large. For example, for the ILD application, the BDW collector required a staggering 3806 instructions per object allocated. The Ultrix libraries only required 63 instructions per object for the same workload. In consequence, the overhead for the Ultrix library was 0.6% and the overhead for the BDW collector was 69.3%, barely leaving any cycles for application to do work in. Yet despite this difference, the performance of the BDW was only 2% worse. For the xfig application, the Ultrix version spent 3.2 % of its instructions on memory management, while the BDW version chewed up 34.5% of its cycles managing memory. Yet in this case the BDW version actually outperformed the Ultrix version by 4%.

When assessing the impact of garbage collection, it is unhelpful to count the number of *collect* events, or even to count the total number of *tock* events which happen in synchronisation with the garbage collector. If response time is the system of property of interest, the performance criteria must be the number of *tock* events which happen between the beginning and end events of application transactions. If throughput is more relevant, the performance criteria will be the number of transaction end events.

Application	Relative overhead (%)	Relative time (%)
SIS	845	153
GEODESY	306	101
ILD	6183	102
PERL	359	126
XFIG	1078	96
GHOST	913	145
MAKE	550	111
ESPRESSO	464	115
PTC	894	113
GAWK	482	157
CFRAC	170	114

Table 6.5: The relationship between memory management overhead and application performance in the Ultrix collector and Boehm-Demers-Weiser collector. All figures derived from those provided by Detlefs et al. [31]. Both the relative instruction overhead of the memory management and the time taken are the figures for the Boehm-Demers-Weiser collector presented relative to the figures for the Ultrix memory management library; in the case when the two are equal, the figure would be 100.

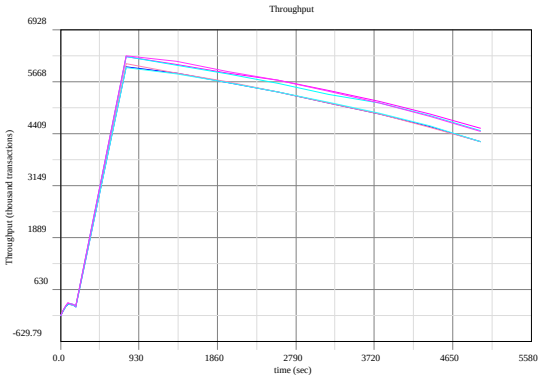
6.3 Workload

While many popular benchmarks use a fixed number of transactions, some, like SPECjbb and SPECjAppServer, use a fixed time instead. This variable workload can create interesting interactions with the garbage collector and make evaluating the effect of garbage collector changes non-trivial.

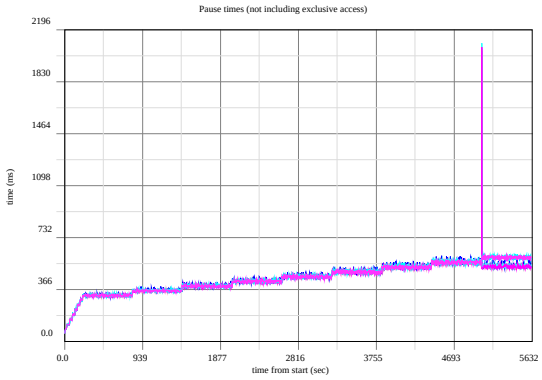
An example of improved garbage collector performance creating improved application performance which causes apparently worsened garbage collector performance is shown in figure 6.10. The thread-local heap size was varied to detune the java virtual machine. When the thread-local heap was small, there was more contention between threads to allocate objects. This illustrates one of the ways garbage collectors can influence performance which is not directly related to garbage collection, as discussed in chapter 3. This contention reduced the application performance and produced generally reduced throughput scores. Unfortunately the example is not perfect, because statistical variations in when compactions were triggered, and the increased fragmentation seen with the smaller thread-local heaps, meant that compactions were triggered in some of the lower-performing runs. These compactions dramatically boosted the throughput in the later warehouses and therefore increased the mean score in a somewhat unrepresentative way. (The fragmentation was more pronounced with the smaller thread local-heaps because objects created at the same time have a tendency to die at the same time, and with the smaller local heaps, objects created at the same time were scattered through the main heap.)

An apparent solution to the feedback problem is to use an 'efficiency'-style metric [55]. The efficiency metric measures the amount of data cleared as a function of pause time. While this eliminates the impact of the increased throughput from the metric, eliminating throughput from a performance measurement is rarely sensible. The efficiency metric is particularly unhelpful since pause times are often positively correlated to application throughput, rather than inversely correlated as might be expected (and as the efficiency metric implicitly assumes). Pause times are not even particularly correlated with response times, as discussed in chapter 4.

A more reasonable solution is to eliminate the feedback by changing the experimental conditions. While SPECjbb is the standard benchmark for the evaluation of large scale Java performance [16], many papers [18, 48, 51, 41, 49, 90] use a fixed-workload variant of SPECjbb called pseudojbb. One of the reasons quoted for using pseudojbb is that most benchmarks give a time result, and using a fixed workload makes the pseudojbb results dimensionally consistent with other benchmarks [18]. However, an important side-effect of using a fixed workload is that the complex feedback between garbage collection performance and application performance is simplified [16]. The workload can



(a) Reported throughput. The jumps in throughput at high warehouse numbers are a result of compactions in the lower-performing runs.



(b) Pause times. There is no significant variation in the mean pause, although the total pause times were larger with bigger thread-local heap sizings.

Figure 6.10: Pause times and reported throughput for SPECjbb2005 runs on a virtual machine which had the thread-local heap size deliberately reduced from the optimum values. The pauses and throughputs are too close to one another to make a legend meaningful; the results are summarised in table 6.6.

TLH size (bytes)	Mean pause (ms)	Total pause (s)	Throughput (ktransactions)
1024	458	1085	3031
1536	456	1080	3037
2048	460	1091	3032
2560	456	1128	3201
3072	471	1224	3082
3584	469	1226	3101

Table 6.6: The relationship between mean pause, total pause time, and mean throughput for SPECjbb2005 runs. The thread-local heap (TLH) size was adjusted to detune the VM and produce the throughput variations. While the correlation is not perfect, mostly because of differences in when compactions were triggered, increased TLH size is associated with improved throughput, and that improved throughput is associated with increased total pause time. The throughput and pauses are plotted in figure 6.10.

never increase or decrease and affect GC characteristics.

The pseudojbb benchmark can certainly simplify the interpretation of benchmark results, but the choice of fixed number of transactions rather than fixed time is not obviously more natural. Benchmarks are intended to give insight into how an application will perform in the field. In some situations, such as a one-time document rendering task, the time taken to process a set workload is clearly more relevant. In others, such as a web server under heavy load, the number of transactions processed during a defined peak period will be the property of interest. Therefore the interaction between the garbage collector and the amount of garbage created cannot be dismissed as simply an artefact of the SPECjbb benchmark which is irrelevant to real application performance.

In chapter 5 it was mentioned that many real world applications do not have clear performance metrics or any facility for providing performance feedback. For applications under steady load, the positive relationship between garbage collection and performance can be used as an informal performance measure. Running the application for a set period of time, or viewing only a time-delimited section of the verbose gc logs and counting the number of collections can provide a surprisingly accurate performance indicator.

TLH size (bytes)	Mean throughput (ktransactions)	Collection count	Log size (kb)
1024	3031	2364	2443
1536	3037	2370	2450
2048	3032	2370	2450
2560	3201	2474	2557
3072	3082	2599	2685
3584	3101	2614	2787

Table 6.7: Log size and performance are often closely correlated. File sizes for the runs plotted in figure 6.10. The collection count and log size are perfectly correlated, while the collection count and throughput share general trends.

Chapter 7

Garbage collection has a disastrous performance impact

The best performance improvement is the transition from the nonworking state to the working state.

John Ousterhost

Because it introduces measurable, and sometimes severe, pauses, it is widely held that garbage collection destroys application performance. While garbage collection does usually have a performance impact, it is often much less severe than is initially apparent. In some cases garbage collection may even improve application performance.

Intuitively, the work of de-allocating objects must be handled by the applications when it is not handled by a virtualisation layer. In complex applications, tracking which objects are in use and which are not, as objects are passed between components, can incur a significant overhead, both in terms of cycles (as methods communicate with one another to trigger object disposal) and memory (since objects often clone and then deallocate an object rather than risk deallocating a shared object which something else was still using).

Memory consumption is another important aspect of the overall performance of an application. While garbage collection generally requires a significant space overhead, this is partially offset by the fact that memory leaks are much more of a problem with unmanaged runtimes. Of course, memory leaks are also a risk with garbage collected languages. Ironically, attempts to minimise the performance impact of garbage collection are a leading cause of memory leaks in garbage collected languages.

7.1 Perceptions of garbage collection performance

There has been widespread public debate on the performance impact of garbage collection. It is clear that at least some of the perceived performance cost of garbage collection is due to the fact that garbage collection is an isolated and relatively visible component.

Consequently, a popular solution to garbage collection performance problems is simply to reduce the visibility of the garbage collector. The documentation for the CMUCL Common Lisp garbage collector suggests:

> The variable ext:*gc-verbose* can be used to disable the status messages that are printed by the garbage collector. This may give the illusion of reducing GC overhead. [62]

The emacs text editor uses a garbage collector. Historically the garbage collector displayed a message when collecting. The garbage collector was widely perceived to be a serious performance bottleneck. The solution to the issue, surprisingly, turned out to be removing the messages, rather than improving the collector.

Erik Naggum, a lisp advocate and emacs contributor, explained the reasoning behind the change to the comp.emacs newsgroup as being squarely one of perception:

> You get the garbage collection messages because Emacs' author thinks it's better to explain why Emacs pauses. However, the problem with this is that a usually invisible overhead in programs that dynamically allocated memory is suddenly becoming visible to the user. C's malloc/free takes time, too, but you never get to see how much it takes because it's not reporting it. Emacs 19.29 will have a variable to suppress those messages, and I can tell you that it's helped me ignore the garbage collection completely, esepecially as I don't notice the delay until it's over, and frequently not even then, because I don't get a message telling me it's about to happen. out of sight, out of mind, virtually. [63]

Some years later, he confirmed that the perception of performance improvement was not restricted to himself:

> Emacs used to tell people that it was garbage collecting. A lot of people complained. So I removed the message and let users at the U of Oslo CS Dept use the modified Emacs. Several people thought that Emacs had become more responsive (*snicker*) and nobody complained about the gc pauses that were no longer announced. [64]

Although it's tempting to assume Naggum is inflating the effect of his change, his assertion is corroborated later in the same thread, when a reader responds

> Ah. so YOU're responsible for this! I wondered were [sic] all those messages went... I thought maybe they took the garbage collection out of emacs. [71]

While the discussion of the performance impact of garbage collection in emacs and lisp is mostly amusing, some debates have been much less light-hearted. In particular, attempts to introduce garbage collection to languages which traditionally use manual memory management have been strongly critised on performance grounds. Collectors exist for C and C++ [98], but these collectors exist strictly as independent libraries, and proposals to include them as part of the core language are generally dismissed.

While there are a number of technical obstacles which make garbage collection in C and C++ less safe and less efficient than implementations which are designed into languages and have full compiler support, the main reason cited for avoiding garbage collection in C and C++ is performance. For example, in a thread on the comp.lang.c++ usenet group, Claudio Puviani concisely respondes to a request for garbage collection support with the statement that "garbage collection is (a) slow, and (b) non-deterministic"[75]. The second point about non-determinism is a reference to the debate over how system resources which are disposed of in finaliser calls are to be handled if the finalisers are called by the garbage collector.

When language vendors announce the introduction of garbage collection support the reaction is often equally hostile, even though the compiler-level support eliminates many of the technical difficulties which make garbage collection a less attractive prospect for C and C++. For example, Borland's Delphi .Net also faced criticism for including garbage collection. In a usenet thread entitled "Make Garbage Collection Optional Then I Will Shut Up" [88] a poster writes, apparently without irony,

> Yeah, just make the garbage collection optional then nobody anywhere will ever have anything to complain about. I remember when people use to liken Borland to Burger King's slogan "Have it your way". I could possibly, just maybe live with the *current* bytecode performance hit, but the garbage collection performance is just way too much.

In the slashdot discussion of Objective C's newly introduced garbage collection [47], a poster called treak007 comments

> I see way too many programmers running around, writing the most ineffective code possible, and relying on fast machines to hide the fact that their code uses 1 gig of ram to display "hello world". That being said, I think that programmers should try to strive for the most effective coding strategy possible, simply because it is good practice [...] The

> main problem I think with Java's memory management is that it doesn't let you control enough. It allows the JRE to control too much memory allocation. That is one of the main reasons that people think that Java is slow.

That is, he argues there is a direct correlation between a lack of control over the memory management and a lack of speed in the application.

The misperceptions about garbage collection extend even to more formal works. In an unpublished paper titled "The Toll of Garbage Collection" Carsten Frigaard attempts to extend Prechelt's language comparison results [74] to C#. He compares the performance of C++ and C# for a phonecode application and concludes that the C# application has performance 37 times slower than that of the C++ application. He also concludes that it consumes four times more memory. His performance figures are dubious for a number of reasons. His descriptions of the challenges he faced porting his C++ code to C# do not inspire confidence in the efficiency of his C# implementation. More fundamentally, he apparently conducts only a single run of his experiment, he does no warm-up run to warm the caches (and JIT, in the C# case) and the total duration of his experiment is less than 1 second in the C++ case and less than 35 seconds in the C# case. The extra initialisation costs of a managed runtime could easily dominate this 35 second period. Many of these are also concerns in the original Prechelt experiment, but Prechelt's description of his scope is more constrained, and he does not attempt to generalise his conclusions to a broad statement on Java performance.

In apparent ignorance of any factors which might affect the universality of his performance results, Frigaard cheerfully attributes the entire performance difference to the impact of the garbage collector. He points out prolonged periods of constant heap size in the memory profile of the C# application and concludes that they are periods of program inactivity during which the garbage collector is running. He neglects to confirm this hypothesis by using the CLR profiler to measure the actual time spent in garbage collection. This is particularly unfortunate since heap size (as opposed to usage) tends to be constant in garbage collected applications during productive periods.

Frigaard moves on to a brief complexity analysis of allocation and deallocation in the two languages. He measured allocation cost and concludes that it is $O(1)$ for C++ and $O(S)$ for C#, where S is the object size. He then produces theoretical deallocation costs of $O(1)$ for C++ and $O(N + \log M + M)$ for C#, where M is the number of allocated objects. This figure is quoted per object, and translates to a complexity for an entire collection of $O(\bar{N}M + M \log M + M^2)$. This is considerably larger than the conventionally quoted figure $O(\bar{N}M)$ [55]. While it is unlikely Frigaard's paper will have much impact, it does illustrate the willingness to attribute untoward performance effects to garbage collection before thorough investigation of other factors has been done.

The association between garbage collection and reduced performance is so strong that there has even been research into introducing explicit memory management to Java. Most notably, Paller implemented explicit deallocation in Java virtual machines [69]. His results apparently justify the attempt, since he did report performance improvements by doing so. His focus was embedded devices, which have small memory capacity and which therefore cannot afford a large memory overhead to improve garbage collection. Paller does not propose that Java programmers should themselves deallocate their objects; instead a static analysis of the bytecode would be used to insert appropriate deallocation instructions. (A similar idea was presented, but not prototyped, by Barth [14].) No deallocation was done for objects which did not have well-defined lifetimes, either because they were unusually long-lived, or because they were shared between threads asynchronously.

Another attempt to partially excise garbage collection from Java was made in the Real-Time Specification for Java [21]. The specification reassigns control for some aspects of memory management back to the developer. The reason given is that

> Garbage-collected memory heaps have always been considered an obstacle to realtime programming due to the unpredictable latencies introduced by the garbage collector. [21]

The real-time Java specification mitigates the obstacle by enabling the programmer to dispense with garbage collection. That is, it was designed to 'allow the allocation and reclamation of objects outside of any interference by any GC algorithm. [21]' Very short-lived and very long-lived objects can be flagged as such by the programmer and allocated into dedicated scoped or immortal heaps.

References between objects in these special memory regions and the normal heap are restricted. The new memory areas are not garbage-collected. Realtime-critical threads may only allocate and manipulate objects from these new memory areas. Because the realtime threads do not have access to objects in the main heap, they may safely pre-empt any conventional garbage collection threads manipulating that heap.

Bacon argues that both the programming model and performance impact of the RTSJ are unacceptable. He further argues that with the right collector (namely his), it is possible to achieve real-time targets without bypassing garbage collection [9].

7.2 Costs

Many of the arguments about the performance cost of garbage collection implicitly assume that garbage collection is necessarily costly and manual memory managegement is necessarily cost-free. Neither of these assumptions stand up to inspection.

For example, the cost of copying collectors is proportional to the amount of live data, rather than the heap size. In the optimimum case, when almost all objects die between collections, copying collectors have costs per object freed of less than one instruction Free calls cannot approach this limit. Detlefs et al measured the costs of the free calls in four popular malloc/free libraries, for eleven applications [31]. The fastes freeing was the G++ library, with a constant cost of 8 instructions per call to free(). One of the other libraries had a constant cost of 18 instructions per call, while two others had variable costs ranging between 28 instructions and 113 instructions, depending on the library and the application.

None of the malloc/free routines were cost-free. The most efficient library was the G++ library, which used 6.32% of the instructions over all applications. The Ultrix library was similar, using 6.97% of the instructions, but it was much more sensitive to the application characteristics. For the CFRAC factoring program, it used 20% of the instructions. The Gnu' library used 17.29% of the instructions on average, but managed to use 52.3% of the instructions when running CFRAC. The very poor performance for CFRAC is surprising, since CFRAC seems to have relatively average characteristics in terms of the number and size of objects allocated, and also in terms of its allocation rate. For comparison, the Boehm-Demers-Weiser collector consumed 36.4% of the total instructions.

7.3 Measuring the performance impact

Neither arguments about optimum cases nor instruction count measurements are particularly satisfactory as a way of assessing the validity of complaints about the performance impact of garbage collection. Optimum cases may never occur, and instruction counts often bear little correlation to actual performance impacts, for the reasons discussed in chapter 6.

Are Paller's results that Java would be better off without garbage collection universally applicable? Measurement of the impact of garbage collection is non-trivial, for a number of reasons. Fully like-for-like comparison is difficult because C and C++ are the only common languages which allow for both garbage collection and manual memory management. C collectors must operate in a hostile environment and cannot take advantage of many of the advantages offered by moving collectors, such as object re-ordering. Unless applications are entirely rewritten, collected applications have to carry the cost of the somewhat strained object manipulations required to avoid object ownership ambiguities, even though there is no need for these patterns in the presence of garbage collection. Cross-language comparisons are even more risky, since languages differ in many respects, not just their memory management strategies. For example, in short runs Java will suffer from high start-up costs, some code paths running interpreted, and also from the overhead of the just-in-time compiler. In long runs the just-in-time compiler may give the advantage to Java, since it has access to dynamic information which can let it make more powerful optimisations than a static compiler. In both of these scenarios, a comparison between C and Java is unlikely to show only differences between the memory management policies of the two languages.

7.3.1 Same-language comparisons

Zorn [98] performed several experiments measuring the performance impact of the widely used Boehm-Demers-Weiser collector for C. In his preamble, he explains

> A major motivation for this measurement is the belief that programmers have misperceptions about the relative performance of different dynamic storage management algorithms. In particular, there is a widespread belief that garbage collection algorithms are both slow and memory-intensive and that there is little or no cost to explicit storage management.

Zorn found the garbage collected programs used 30 to 150% more memory than the versions with manual memory management. Largely because the diffusion of the live data across a greater memory space resulted in more page faults and cache misses, Zorn infers the performance of the garbage collected programs is worse, but does not provide specific figures. This oversight is corrected by Detlefs, Dosser, and Zorn in their expanded version of the same experiment [31]. They measured the total program execution time across eleven programs to be 21% worse on average. In the worst case garbage collection added 53% to the execution time, while in the best case the garbage collected version of a program outperformed the manually managed ones by between 4% and 9% (depending on which allocator it is compared to).

Zorn's experiment did not alter the code except to replace calls to malloc with calls to GC_malloc and null out calls to free. This means that all the extra in-code infrastructure required by manual memory management (such as object cloning) [94] remained. Furthermore, the Boehm-Demers-Weiser collector is unable to take advantage of many of the performance enhancements available with moving collectors, such as compaction or copying to improve locality. Jones [55] suggests his results are at most an upper bound on the performance cost of garbage collection.

Hertz and Berger performed a clever experiment in which they traced application execution with a profiler in order to extract detailed object reachability traces [48]. This trace was then used to produce an instrumented version of the application which explicitly deallocates objects at the appropriate time. They test eight benchmarks, including a fixed-workload variant of SPECjbb, most of the SPECjvm suite, and an XML database system. Using an Appel style generational collector with non-copying mature space collector they were able to match the performance of explicit memory management, and sometimes even exceed the explicit memory management performance by up to 9%. Achieving this performance required a much more liberal heap allocation than that required by the explicit memory management, however.

A weakness of their experiment is that the final comparison is produced by running the instrumented code and uninstrumented code in a simulator. Any deviations in behaviour between the simulator and a physical system could affect the validity of the comparison. A more fundamental problem is the converse of the issues faced by Zorn; because the applications they tested were coded to be run with automatic memory management, they did not include any of the performance-damaging defensive protocols for managing ownership of shared objects. This problem seems unavoidable in any comparison between explicit and automatic memory management. If applications were recoded before comparison to take advantage of the neater coding style offered by automatic memory management, it would be difficult to avoid inadertently introducing other optimisations while still making all memory-management-related optimisations. It would be even more difficult to *prove* that no other optimisations had been introduced which might invalidate the experiment.

7.3.2 Cross-language comparisons

Blackburn et al [17] compare the MMTk memory management toolkit for the Jikes RVM with runs of the same micro-benchmark implemented in C. The micro-benchmark constructs a simple binary tree with two reference fields and two data fields. They use the GNU C librarys malloc routines; the performance of other C libraries will vary. Even when the workload is the same, working out the fairest comparison is non-trivial. The GNU malloc uses a function call, and so to be balanced inlining should be disabled in MMTk. This reduces the performance of MMTk by about 35% to 45%. Conversely, Java returns zeroed memory and malloc does not. The allocation call calloc does return

zeroed memory, and so for full equivalence calloc rather than malloc should be used. Switching to calloc and zeroing memory does have a significant cost (29%). It is also not clear that using calloc is fair, since garbage collected systems must zero memory on allocation to avoid the risk of false pointers in the newly allocated data [55], while C does not have this requirement. The closest comparison (MMTk MarkSweep - noinline versus C calloc) shows that C has a small advantage 6.8%. On the other hand, the comparison of the optimum configurations, which is arguably fairer, gives Java a slight advantage instead.

While this seems to be a very positive result for garbage collection, it must be remembered that Blackburn et al only ported one micro-benchmark.

Even a casual inspection of the results of Zorn and Detlefs shows that the relative performance of allocators and collectors is highly sensitive to the application, and general statements cannot be made on the basis of one application. This is even more true when that application is a micro-benchmark; their limited codebase and tight scope make them particularly unsuitable as a basis for broad performance generalisations.

A more realistic workload is used by Vivanco and Pizzi in their comparison of Java and C++ for an fMRI processing application [91]. They implemented two data processing algorithms in both Java and C++ and then compare them across two C++ compilers and six JVMs. Both the data processing algorithms rely heavily on array access and mathematical manipulations.

For their simpler fMRI processing application, they find that the fastest Java implementation is three times slower than the C++ version. The newer 1.3 and 1.4 JVMs also significantly outperform 1.2-era JVMs. However, the execution times for almost all versions are under a second. On this kind of time scale startup and JIT costs easily dominate the execution time. Startup and JIT costs would certainly outweigh any garbage collection in terms of performance impact. Vivanco and Pizzi also test a more involved algorithm whose execution time is of the order of a minute. For this workload the fastest JVM, an IBM 1.3 machine, runs as fast as the pgCC-compiled implementation (29 seconds) and faster then the gcc implemention, which took 31 seconds. The Sun 1.2 JVM took 264 seconds, so it is also clear that the performance of Java virtual machines varied between vendors and had generally improved over time. While it is therefore reasonable to assert that some Java implementations compared very favourably with the C++ implementations, it would certainly not be reasonable to assert that *all* of them do.

The strength of the Vivanco experiment is that the primary interest of Vivanco and Pizzi is processing their fMRI data sets, and so their application is a genuine real-world application. Real-world applications are generally more reliable for performance experiments than synthetic applications, particularly when memory allocation and memory access patterns are part of what is being tested [96]. The weakness of their experiment is that because they are mostly interested in doing real work, their background is not in benchmarking or performance analysis. In consequence some of their methodology is not as rigorous as it could be.

A more standardised comparison between Java and un-garbage-collected languages is offered by the SciMark benchmark, which is available for Java, C, and Fortran [73]. The SciMark benchmark includes five components, each of which performs a numerically-oriented computation. On SPARC processors, C outperforms 1.3 Java JVMs by a factor of roughly 2 over all five components of the SciMark benchmark. On Intel Windows hardware, however, even a 1.2 JVM was able to outperform the C version on all but one component. On Linux, with a different compiler, the C performance was improved and it was broadly similar to the Java performance on all but one component, with Java having the lead in two components. The Monte Carlo integration component performed better with C than it did with Java on all platforms. For Fortran on Power architectures, the Java and Fortran performance is broadly similar, with Java sometimes outperforming Fortran and sometimes not [72].

As with the fMRI processing applications used by Vivanco and Pizzi, the SciMark benchmark focuses heavily on numerical computation rather than more general work. The utility of the scientific benchmarks for isolating effects related to garbage collection is limited since many traditional scientific applications have relatively stable memory requirements and little object churn. [23].

7.4 Space overhead

Both the cross-language and same-language comparisons show that the performance impact of garbage collection, while sometimes observable and significant, is not always significant or even observable. It is clearly smaller than some of the more hysterical popular assessments of garbage collection imply. However, one area where the garbage collection does struggle to match manual memory management is in the space overhead. In general, garbage collection requires significantly more memory than manual memory management, and the performance of the application increases as the space allocated to the collector increases.

This was particularly clear in Hertz and Berger's results. In order to match the performance of manual memory management, they needed five times more memory than manual memory management. With only three times as much memory, their benchmarks ran on average 17% slower with automatic memory management as explicit memory management. When the heap was reduced to twice the minumum, the performance was degraded by 70%. They observe that when physical memory is scarce paging in large heaps causes garbage collection to run an order of magnitude slower than explicit memory management.

Hertz's results should not be taken as universal; the relationship between heap size and performance is highly implementation and application specific, and larger heaps are not even always correlated with improved performance. An apparently dramatic counter-example is provided by Blackburn et al. Blackburn et al. [18] measured the relationship between heap size and performance for an Appel-style generational collector. They appeared to find that the optimum heap size relative to the minimum memory requirement was highly dependent on the benchmark used. For pseudojbb, the optimum heap size was only about 1.75 of the minimum. For both smaller and larger heaps the performance degraded sharply by about 50%. For the SPECjvm benchmarks, near optimum performance was achieved in most cases for heaps around twice the minimum. Increasing the heap size beyond that point gave improvements or degradations of only a few percent. They obtained similar results for an Appel-style generational collector. The dramatic degradation of pseudojbb seems surprising, and Blackburn et al. offer no explanation for it. However, close examination of the hardware description clarifies the situation. The minimum heap requirement for the pseudojbb run was 70 MB, about twice the minimum requirement for any of the other benchmarks. The machine they used for their experiments only had 128 MB RAM, so paging was guaranteed as soon as the heap exceeded 1.8 times the minimum size. Collector access to memory which is paged out is significantly slower than access to normal memory. This is exactly the degradation in large heaps discussed by Hertz and Berger. Somewhat oddly Blackburn et al do not include this explanation in their analysis of the results.

Even when the pseudojbb results are discounted, however, Blackburn et al. show much less benefit from large heaps than Hertz does. The six SPECjvm benchmarks they tested all had heap requirements small enough that there should be little danger of paging. They all achieved performance that was within 5% of the best measured performance at heaps two times the minimum required size.

Why are these results so different from Hertz's? The main difference probably has to do with the relative size of the heaps to the caches. The implementations of the garbage collectors may also be different, even though the general class of algorithm is the same.

Comparison of figures 6.5(b) and 6.6(a) illustrates how implementation-specific the sensitivity to heap size is. Doubling the heap gave the mark-sweep collector a significant boost, but only gave the generational collector a marginal benefit. The differences are summarised in table 7.1.

Policy	Heap size (MB)	Occupancy (%)	Mean throughput (ktransactions)	Difference (%)
Mark-sweep	1000	55	3387	19
	2000	29	4036	
Generational	1000	55	3559	1.5
	2000	29	3611	

Table 7.1: The effect of heap size changes on mark sweep and generational collectors.

The tightly contrained heap is also the context in which Pattel's results should be taken.

Shaham et al. define an effectiveness criteria for Java based on the time between when objects are last used, and the time they are collected [80]. In a sense this is just a different way of measuring the heap usage overhead of the collector; collectors with very little space overhead should be collecting objects quite promptly, while collectors with large overheads are by implication allowing objects to linger on well past their useful lifespan. They instrumented a JVM to identify dragged objects - that is, objects which are no longer in use but which have not been collected. These objects represent transient memory leaks. Conservative collectors do not collect all non-live objects, and are therefore less effective by Shaham's criteria. Unfortunately, collectors which collect more frequently are more effective by the same criteria. In constrained heaps, frequent collection is associated with serious performance degradations, so the effectiveness measure is inversely related to the application performance in these cases. More seriously, there is no distinction made between memory leaks caused by conservative collection, and memory leaks caused by poor application implementation. Shaham compares the impact of dragged objects in several benchmarks from the SPECjvm suite, with a single JVM. He uses these results to draw conclusions about the effectiveness of GC in Java, when what he has measured is clearly inter-application differences in dropping references to objects which are no longer required.

As a final note, although garbage collection requires more memory than conventional allocators, application memory requirements are suprisingly variable even when garbage collection is not used [98]. The choice of `malloc/free` library can significantly influence how much memory is consumed.

7.5 Performance workarounds

Misconceptions about the performance of garbage collection have given rise to some misguided performance workarounds.

7.5.1 Object pooling

Object pooling is sometimes presented as a Java best practice in order to minimise the performance impact of the garbage collector [30, 2]. The idea is that object instances are kept in a pool when no longer required and then repopulated with fresh data when a new instance is required. This saves the garbage collector from having to collect the original object and means that the 'creation' of the new object will not involve a request for memory which could trigger an allocation failure and a garbage collection.

There are several problems with this strategy. The infrastructure required to manage the pooling increases the amount work done by the application. If object instances are shared, working out when an object may be safely returned to the pool is non-trivial and may involve many of the expensive and awkward management of inter-component references discussed in chapter 2. To add insult to injury, for copying collectors, the effort required to do a collection is proportional to the amount of live data, rather than the size of the heap, and so holding onto unused objects will actually increase the duration of the garbage collection pauses.

In order to ensure an adequate supply of objects, there may be many unused instances held in the pool at any one time. In some designs [30], it is possible for an arbitrarily large number of objects to be added to the pool and then never removed or re-used. To avoid this memory leak it is necessary to run an independent clean-up thread as well. If the pooled objects hold references to other objects and these references are not cleared before returning the object to the pool, the memory impact of pooling the object can be much greater than simply the size of the object itself. With densely populated reference structures, the leaked memory may require the garbage collector to either collect much more than would otherwise be required, or expand the heap unnecessarily.

It is perversely fitting that the attempt to reduce the performance impact of garbage collection by returning to a more manual application-level method of memory management reintroduces many of the problems automatic memory management was meant to avoid. In particular, it requires arcane inter-module dependencies and allows potentially serious memory leaks.

Chapter 8

Conclusions

Despite the age of the technology and its vitality as a research area, garbage collection is frequently misunderstood.

The myth persists that garbage collection is only required by the weak of mind. Instead of being regarded as an essential feature of any modern high-level language, garbage collection is seen as a crutch for developers too sloppy to manage to deallocate what they allocate. In fact, managing object ownership in a multi-developer multi-component project is seriously non-trivial. Complicated protocols have been developed to avoid ambiguities in object ownership and their attendant memory leaks and dangling pointer crashes. These techniques interfere with the transparency of the design, slow applications down, and increase application space requirements. They are also not foolproof, and so memory leaks and application crashes may still occur. In contrast, garbage collection allows robust code which naturally expresses the intended design.

Because garbage collection is generally used as a synonym for automatic memory management, it is easy to forget that automatic memory management includes both a collection and an allocation component. It is also easy to forget that manual memory management requires an allocator, and that these allocators are neither homogeneous in performance across implementations, nor cost-free. Because of its ability to compact and rearrange objects, garbage collection allocation can outstrip conventional allocators, both in terms of the speed of the allocation and the speed with which the application can access the allocated objects.

The measurable pauses in application activity introduced by garbage collection are often viewed as an inhibitor to good application response times. Attempts to improve response times often focus on reducing pause times and neglect both the frequency and clumpiness of the pauses, and application throughput. Clustered small pauses are generally no better than single large pauses in terms of application response times. Queueing theory demonstrates that good application throughput is equally important in maintaining good response times.

Pause times are also often used as a metric of general application performance. An increase in pause times is interpreted as a performance degradation. While this is sometimes true, it is critical to clarify which performance properties are of interest before drawing conclusions about what is happening to them. Often an increase in individual pause times is not accompanied by an increase in the total pause time. It also often the cause that an increase in pause times may indicate a worsening of the average-case performance but an improvement to the worst-case performance.

Even total pause time is often uncorrelated to application performance. Concurrent collectors hide collection work in with the normal application work, so that a low reported total pause is sometimes accompanied by a very high actual amount of work. More surprisingly, doing lots of collection work is often perfectly compatible with best-of-breed application performance, while low garbage collection overheads are sometimes accompanied by dismal application performance. In fact, for fixed measurement periods, increases in garbage collector performance which cause improved application performance can cause increased garbage collection overhead. That is, the apparently worsened garbage

collector performance is a direct and inevitable consequence of the genuinely improved collector performance.

There is a widespread consensus that the general performance impact of garbage collection is deleterious and large. However, assessing the performance of garbage collection is non-trivial, and judging the relative performance of very different systems is a particularly murky art. There is little evidence about the performance impact of garbage collection compared to manual memory management, but the existing experiments show that garbage collection usually has a relatively minor performance impact, and in some cases garbage collected systems outperform those with manual memory management. This performance usually comes at the cost of much greater memory requirements.

Bibliography

[1] Erwin S. Andreasen and Henner Zeller. *LeakTracer*, 1999.

[2] Lillian Andres. Avoid bothersome garbage collection pauses. *Java Developers Journal*, July 2003.

[3] Anonymous. SASL plugin programmer's guide. http://www.sendmail.org/ ca/email/-cyrus2/plugprog.html.

[4] Anonymous. D language. http://www.skyos.org/board/viewtopic.php?p=91820&sid=e56df2df37ff899da22c8ac November 2005.

[5] Apple. *Memory Management Programming Guide for Core Foundation: Ownership Policy*, October 2006.

[6] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[7] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 269–281, New York, NY, USA, 2003. ACM Press.

[8] David F. Bacon. Bravely using java in the new world of complex real-time systems. www.research.ibm.com/people/d/dfb/talks/Bacon05BravelyTalk.ppt, 2005.

[9] David F. Bacon, Perry Cheng, and V. T. Rajan. The Metronome: A simpler approach to garbage collection in real-time systems. In R. Meersman and Z. Tari, editors, *Proceedings of the OTM Workshops: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume 2889 of *Lecture Notes in Computer Science*, pages 466–478, Catania, Sicily, November 2003. Springer-Verlag.

[10] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.

[11] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[12] Henry G. Baker. The Treadmill: Real-Time Garbage Collection without Motion Sickness. *ACM Sigplan Notices*, 27(3):66–70, March 1992.

[13] Katherine Barabash, Yoav Ossia, and Erez Petrank. Mostly concurrent garbage collection revisited. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 255–268, New York, NY, USA, 2003. ACM Press.

[14] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Commun. ACM*, 20(7):513–518, 1977.

[15] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, MA, 2000.

[16] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, October 2006. ACM Press.

[17] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In Laurie J. Hendren, editor, *Proceedings of PLDI'02 Programming Language Design and Implementation*, pages 153–164, Berlin, June 2002. ACM Press.

[19] Hans-Juergen Boehm. Advantages and disadvantages of conservative garbage collection. http://www.hpl.hp.com/personal/Hans_Boehm/gc/issues.html.

[20] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[21] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, Mark Turnbull, and Rudy Belliardi. Real-time for java specification (JSR 01). http://www.rtsj.org/specjavadoc/book_index.html, 2000.

[22] Sem C. Borst, Onno J. Boxma, John A. Morrison, and R. Núñez Queija. The equivalence between processor sharing and service in random order. *Oper. Res. Lett.*, 31(3):254–262, 2003.

[23] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking java against C and Fortran for scientific applications. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 97–105, New York, NY, USA, 2001. ACM Press.

[24] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 125–136, New York, NY, USA, 2001. ACM Press.

[25] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 46–56, New York, NY, USA, 2005. ACM Press.

[26] John Coomes and Tony Printezis. Garbage collection in the HotSpot (tm) java virtual machine. Technical Report TS-3153, Sun, 2003.

[27] Standard Performance Evaluation Corporation. SPECjbb2000 run and reporting rules. http://www.spec.org/jbb2000/docs/runrules.html, 2000.

[28] Standard Performance Evaluation Corporation. SPECjbb2005 whitepaper. http://www.spec.org/jbb2005/docs/WhitePaper.html, 2005.

[29] Oisin Curtin. Two words produces error. Usenet: borland.public.cppbuilder.language group, February 1998.

[30] Thomas E. Davis. Build your own ObjectPool in Java to boost app speed. *JavaWorld.com*, June 1998.

[31] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6):527–542, 1994.

[32] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 37–48, New York, NY, USA, 2004. ACM Press.

[33] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 269–280, Washington, DC, USA, 2006. IEEE Computer Society.

[34] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 69–80, New York, NY, USA, 2003. ACM Press.

[35] Ralph L. Disney and Dieter König. Queueing networks: a survey of their random processes. *SIAM Review*, 27(3):335–403, September 1985.

[36] Nate Eldredge. *YAMD, Yet Another Malloc Debugger*, 2001.

[37] David Eng. Garbage collection. Usenet: comp.lang.c++.moderated group, May 2005.

[38] Cal Erickson. Memory leak detection in C and C++. *Linux J.*, 2002(101):9, 2002.

[39] Cal Erickson. Memory leak detection in embedded systems. *Linux J.*, 2002(101):9, 2002.

[40] David Evans. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 44–53, New York, NY, USA, 1996. ACM Press.

[41] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Method-level phase behavior in java workloads. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 270–287, New York, NY, USA, 2004. ACM Press.

[42] Paul Graham. Java's cover. http://www.paulgraham.com/javacover.html, April 2001.

[43] Donald Gross and Carl M Harris. *Fundamentals of Queueing Theory.* John Wiley and Sons, February 1998.

[44] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In USENIX, editor, *Proceedings of the Winter 1992 USENIX Conference: January 20 — January 24, 1992, San Francisco, California*, pages 125–138, Berkeley, CA, USA, Winter 1992. USENIX.

[45] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 156–164, New York, NY, USA, 2004. ACM Press.

[46] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 168–181, June 2003.

[47] William Henderson. Xcode update gives Objective-C garbage collection. http://it.slashdot.org/article.pl?sid=06/08/07/2126253, August 2006.

[48] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 313–326, New York, NY, USA, 2005. ACM Press.

[49] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanovic. Generating object lifetime traces with merlin. *ACM Trans. Program. Lang. Syst.*, 28(3):476–516, 2006.

[50] Matthew Hertz, Neil Immerman, and J. Eliot B. Moss. Framework for analyzing garbage collection. In *TCS '02: Proceedings of the IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science*, pages 230–242, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[51] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 359–373, New York, NY, USA, 2003. ACM Press.

[52] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 36–49, New York, NY, USA, 2002. ACM Press.

[53] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. *SIGPLAN Not.*, 39(10):69–80, 2004.

[54] Watts S. Humphrey and James W. Over. The personal software process (PSP): a full-day tutorial. In *ICSE '97: Proceedings of the 19th international conference on software engineering*, pages 645–646, New York, NY, USA, 1997. ACM Press.

[55] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 1997 (twice), 1999, 2000, 2003, and 2004.

[56] Martin Karlsson, Kevin E. Moore, Erik Hagersten, and David A. Wood. Memory system behavior of Java-based middleware. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 217, Washington, DC, USA, 2003. IEEE Computer Society.

[57] Kelton. outproc exe - BSTR memory management. Usenet: microsoft.public.vc.atl group, July 2001.

[58] Haim Kermany and Erez Petrank. The compressor: concurrent, incremental, and parallel compaction. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 354–363, New York, NY, USA, 2006. ACM Press.

[59] David L. Levine, Christopher D. Gill, and Douglas C. Schmidt. *Object lifetime manager a complementary pattern for controlling object creation and destruction*, pages 495–534. Cambridge University Press, New York, NY, USA, 2001.

[60] Frank Levine. *TPROF*.

[61] John Machin. How properly manage memory of this PyObject* array? Usenet: comp.lang.python group, November 2006.

[62] Eric Marsden. Cmucl: Understanding the garbage collector. Technical report, CMUCL, August 2004.

[63] Erik Naggum. GNUS: Why does it add newsgroups I do not want? Usenet: comp.emacs group, January 1995.

[64] Erik Naggum. Merriam-Webster's Collegiate Encyclopedia. Usenet: comp.lang.lisp group, November 2001.

[65] Juha Niemenen. Microsoft compares programming languages. http://warp.povusers.org/MicrosoftComparingLanguages/, 2003.

[66] JKop (no other name available). C++ sucks for games. Usenet: comp.lang.lisp group., October 2004.

[67] Le Lapin Chaud (no other name available). Garbage collection. Usenet: comp.lang.c++.moderated group, May 2005.

[68] Max (no other name available). Are Paul Graham and Joel Spolsky right? or: Should I start my own software company? http://ex-mentis.blogspot.com/2006/05/are-paul-graham-and-joel-spolsky-right.html, May 2006.

[69] Gabor Paller. Increasing Java performance in memory-constrained environments using explicit memory deallocation. Presented at Net.ObjectDays and published at http://pallergabor.uw.hu/common/mac_increasing_java.pdf, September 2005.

[70] Geoffrey Phipps. Comparing observed bug and productivity rates for Java and C++. *Softw. Pract. Exper.*, 29(4):345–358, 1999.

[71] Alain Picard. Merriam-Webster's Collegiate Encyclopedia. Usenet: comp.lang.lisp group, November 2001.

[72] Roldan Pozo. Java and scientific computing. In *Joint Workshop of the Organisation Associtave du Parallelisme (ORAP), and the Swiss Forum for High-Performance Computing (SPEEDUP)*, October 2001.

[73] Roldan Pozo and Bruce R Miller. About scimark 2.0. Technical report, National Institute of Standards and Technology, March 2004.

[74] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[75] Claudio Puviani. dtor questions & garbage collection. Usenet: comp.lang.c++ group, December 2002.

[76] Rudesindo Núñez Queija. Sojourn times in a processor sharing queue with service interruptions. *Queueing Syst. Theory Appl.*, 34(1-4):351–386, 2000.

[77] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked concurrent language. Technical Report CSL-84-7, Xerox PARC, July 1985.

[78] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. MC^2: high-performance garbage collection for memory-constrained environments. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 81–98, New York, NY, USA, 2004. ACM Press.

[79] Peter Schröder. Style guide for C and C++ programming, 2005.

[80] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. On effectiveness of GC in Java. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 12–17, New York, NY, USA, 2000. ACM Press.

[81] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of java applications at allocation and garbage collection times. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 13–25, New York, NY, USA, 2002. ACM Press.

[82] David Siegwart and Martin Hirzel. Improving locality with parallel hierarchical copying GC. In Erez Petrank and J. Eliot B. Moss, editors, *ISMM*, pages 52–63. ACM, 2006.

[83] Anthony J. H. Simons. Borrow, copy or steal?: loans and larceny in the orthodox canonical form. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 65–83, New York, NY, USA, 1998. ACM Press.

[84] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, 1991.

[85] Hideaki Takagi. Queue with priorities. In Michiel Hazewinkel, editor, *Encyclopaedia of Mathematics*. Springer-Verlag, 2001.

[86] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture*, pages 533–544, New York, NY, USA, 1998. ACM Press.

[87] J. L. van den Berg and O. J. Boxma. The M/G/1 queue with processor sharing and its relation to a feedback queue. *Queueing Syst. Theory Appl.*, 9(4):365–402, 1991.

[88] Mike Vance. Make garbage collection optional then I will shut up. Usenet: borland.public.delphi.non-technical group, June 2004.

[89] Martin T. Vechev, Eran Yahav, and David F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 341–353, New York, NY, USA, 2006. ACM Press.

[90] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Space-efficient 64-bit java objects through selective typed virtual addressing. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 76–86, Washington, DC, USA, 2006. IEEE Computer Society.

[91] Rodrigo Vivanco and Nicolino Pizzi. Computational performance of Java and C++ in processing fMRI datasets. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN Conference on object-oriented programming, systems, languages, and applications*, pages 100–101, New York, NY, USA, 2002. ACM Press.

[92] Andy Wharmby. Personal communication, July 2006.

[93] Harrison White and Lee S. Christie. Queuing with preemptive priorities or with breakdown. *Operations Research*, 6(1):79 – 95, January 1958.

[94] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *IWMM*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer, 1992.

[95] Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994.

[96] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[97] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference*, pages 115–125, New York, NY, USA, 2005. ACM Press.

[98] Benjamin G. Zorn. The measured cost of conservative garbage collection. *Software - Practice and Experience*, 23(7):733–756, 1993.

[99] Benjamin G. Zorn and Paul N. Hilfinger. A memory allocation profiler for C and Lisp programs. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1988.

End of Document