LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-581

CACHE PERFORMANCE OF GARBAGE-COLLECTED PROGRAMMING LANGUAGES

Mark B. Reinhold

September 1993

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139

Cache Performance of Garbage-Collected Programming Languages

Mark B. Reinhold

Technical Report 581
September 1993

This report is a revised version of the author's doctoral dissertation of the same title, which was supervised by Professor John V. Guttag and submitted to the Department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology in August 1993.

This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contracts N00014-89-J-1988 and N00014-92-J-1795.

Author's current address: NEC Research Institute
Four Independence Way
Princeton, NJ 08540
mbr@research.nj.nec.com

Abstract

As processor speeds continue to improve relative to main-memory access times, cache performance is becoming an increasingly important component of program performance. Prior work on the cache performance of garbage-collected programming languages has either assumed or argued that conventional garbage-collection methods will yield poor performance, and has therefore concentrated on new collection algorithms designed specifically to improve cache-level reference locality. This dissertation argues to the contrary: Many programs written in garbage-collected languages are naturally well-suited to the direct-mapped caches typically found in modern computer systems.

Using a trace-driven cache simulator and other analysis tools, five nontrivial, long-running Scheme programs are studied. A control experiment shows that the programs have excellent cache performance without any garbage collection at all. A second experiment indicates that the programs will perform well with a simple and infrequently-run generational compacting collector.

An analysis of the test programs' memory usage patterns reveals that the mostly-functional programming style typically used in Scheme programs, in combination with simple linear storage allocation, causes most data objects to be dispersed in time and space so that references to them cause little cache interference. From this it follows that other Scheme programs, and programs written in similar styles in different languages, should perform well with a simple generational compacting collector; sophisticated collectors intended to improve cache performance are unlikely to be effective. The analysis also suggests that, as locality becomes ever more important to program performance, programs written in garbage-collected languages may turn out to have a significant performance advantage over programs written in more conventional languages.

Key words and phrases: Cache memories, dynamic storage management, garbage collection, programming-language implementation, Scheme.

To my mother, who taught me to learn;
to my father, who taught me to work.

Acknowledgements

Without John Guttag's enthusiasm, encouragement, and advice, my graduate education likely would have ended five years and one degree earlier. John frequently saved me from sinking in a sea of too-detailed thoughts, usually by reminding me to think about the big picture but, at least once, by simply insisting that it was time to start writing. He brought the keen eye of an English major to my prose, and proved an invaluable coach in the hunt for a research job. I learned much of what I know about garbage collection while trying to explain it all to John; I hope that I managed to teach him half as much about garbage collection as he taught me about doing research.

Bert Halstead and Butler Lampson were ideal thesis readers, asking incisive questions at just the right times. Helpful comments came from many others, including Alan Bawden, Mark Day, Steve Garland, Daniel Jackson, David Kranz, Scott Nettles, Nate Osgood, Jim O'Toole, Tim Shepard, Ellen Spertus, Raymie Stata, Yang-Meng Tan, Mark Vandevoorde, Carl Waldspurger, and Jeannette Wing.

Several existing software systems were essential to this work. David Kranz answered questions about the internals of the T system and helped me port ORBIT to big-endian MIPS machines. Josh Guttman spent an afternoon demonstrating IMPS and helped me get it running at MIT. Mark Hill provided his cache simulator, TYCHO, which I used to validate my own cache simulator. Tom Simon contributed the NBODY code.

This document was prepared with Donald Knuth's TEX system, a source of both programming frustration and typographical delight. The graphical style aspires to the standards set by Edward Tufte's wonderful books on visual design; most of the graphs were created with Jim Plank's JGRAPH program.

Finally, I thank my dear friends in Bryn Mawr, for warm company and good movies; my siblings, for making me laugh and making me think; and my parents and grandparents, for giving me life and helping to make this work possible.

Arlington, Massachusetts MBR
September 1993

1. Introduction

A key feature of modern programming languages such as Lisp, Smalltalk, CLU, and ML is an automatically garbage-collected heap. The primary advantages of garbage collection are correctness and productivity: A garbage collector relieves the programmer from having to worry about manual storage deallocation and the associated dangers of dangling pointers and storage leaks. In terms of programming effort, the cost of correctly using manual deallocation is significant. For one language without garbage collection, it has been estimated that programmers spend about 40% of their time solving problems related to manual deallocation [62].

A further advantage of garbage collection is that, in some situations, it can actually improve program performance. For example, garbage collectors have long been used to improve the performance of programs by improving their virtual-memory performance. This is done by designing the collector to move data objects so that most working data is kept in physical memory. The cost of running such a collector is usually smaller than the improvement that is achieved by reducing the program's page faults [23]. In some systems, collectors of this kind are often crucial to good program performance [49, 66].

This dissertation considers the problem of implementing garbage-collected languages in relation to a different part of the memory hierarchy, namely the cache. One of the most significant trends in computer technology involves the relative speeds of processors and main-memory chips: Processors are getting faster, by a factor of 1.5 to 2 per year, while the speed of main-memory chips is improving only slowly [27, 28]. This widening gap has motivated hardware designers to seek improved performance by inserting one or more high-speed cache memories between the processor and the main memory. A cache miss on current high-performance machines costs tens of processor cycles; if the current trend continues, a miss on such machines will soon cost hundreds of cycles. Thus cache performance is becoming an increasingly important component of program performance.

Given that garbage collectors are capable of improving virtual-memory performance by rearranging data objects, it is natural to ask whether they could improve cache performance by similar means. This possibility has been investigated by several researchers, who have studied collectors designed to move data objects so that most working data is kept in the cache [75, 79]. Because caches are so much smaller than main memory, such collectors must be invoked frequently if they are to be effective. While the cost of running such an *aggressive* collector may be significant, the hope is that it will be smaller than the improvement that is achieved by reducing the program's cache misses.

Prior work on the cache performance of garbage-collected programming languages either assumes or argues that programs written in such languages will have poor cache performance if little or no garbage collection is done. In contrast, the primary claim of this dissertation is that many such programs are naturally well-suited to the direct-mapped caches typically found in high-performance computer systems. Complex and costly means for improving cache performance, such as aggressive garbage collection, are unlikely to be either necessary or effective.

Overview. The investigation begins in Chapter 2 with a simulation-based study of five nontrivial, long-running Scheme programs, compiled and run in a high-quality Scheme system. The primary metric of program performance is total running time, as measured in processor cycles. Running time thus includes cycles used to execute instructions as well as cycles in which the processor is stalled, *e.g.*, waiting for a cache miss to be serviced.

First, a control experiment is performed to determine the extent to which the cache performance of these programs can be improved. The experiment shows that the programs have excellent cache performance without any garbage collection at all: They spend less than five percent of their total running time, on average, waiting for cache misses. Improving cache performance hardly seems necessary; no improvement method that imposes significant runtime costs of its own could be effective. Aggressive garbage collection is likely to be one such method.

In practice, limitations on physical memory require that some sort of garbage collection be done in order to ensure good virtual-memory performance. The results of the control experiment suggest that a good collector for the test programs is one that collects infrequently, in order to take advantage of the programs' naturally good cache performance, but frequently enough to minimize virtual-memory page faults. This hypothesis is tested in the second experiment, which shows that, in most cases, the programs perform well with a simple, efficient, and infrequently-run compacting collector. In the remaining cases, they should perform well with a simple and infrequently-run generational compacting collector.

The results of Chapter 2 are limited to just the five test programs. In order to support generalizations to other Scheme programs and to programs in other garbage-collected languages, Chapter 3 establishes a connection between the manner in which the test programs use memory and their measured cache performance. Like many garbage-collected languages, Scheme encourages a mostly-functional style of programming. Two important consequences of this style are that most data objects have very short lifetimes, and most are only referenced a few times. These properties, in combination with the object allocator's linear sweep through memory, cause most objects to be dispersed in time and space so that references

to them cause little cache interference. The few objects for which these properties do not hold are usually referenced in such a way that they more often improve cache performance rather than degrade it. Hence Chapter 3 concludes that the test programs have good cache performance because their memory behaviors are naturally well-suited to direct-mapped caches.

Chapter 4 builds upon the results of Chapter 3. First, three means by which the performance of the test programs might be improved are presented; none of these methods have significant runtime costs. Then it is argued that the behavioral properties leading to good cache performance should hold for other Scheme programs, and are likely to hold for programs in other garbage-collected languages. The conclusions of Chapter 2 and the performance improvements discussed earlier in Chapter 4 are thereby generalized. The chapter closes by conjecturing that garbage-collected languages may have a significant performance advantage over more conventional languages on fast computer systems.

Finally, Chapter 5 summarizes the results, reviews prior work, and discusses topics for future work.

2. Measurements of cache performance

This investigation is based upon studies of the cache performance and memory behavior of five nontrivial Scheme programs. After describing the programs and delimiting the cache design space, this chapter focuses on two cache-performance experiments.

The first is a control experiment. Before considering methods by which cache performance might be improved, it is appropriate to determine how much improvement is possible. This is done by measuring the cache performance of the test programs when run without any garbage collection at all. If this experiment were to show that the programs have poor cache performance without collection, then some method of improving cache performance would be called for.

In fact, the control experiment shows that the opposite is true: When run without garbage collection, the test programs have excellent cache performance. Seeking improved cache performance hardly seems necessary. There is so little room for improvement that no improvement method with significant runtime costs of its own could be effective.

In practice, it is not possible to run programs with an unbounded amount of physical memory, so some sort of garbage collection must be done in order to ensure acceptable virtual-memory performance. The results of the control experiment suggest that a good collector is one that collects rarely, in order to approximate the non-collection case and thereby take advantage of the programs' naturally good cache performance, yet often enough to minimize virtual-memory page faults.

This hypothesis is tested in the second experiment, which measures the cost of running the test programs in a modest amount of memory with a simple, efficient, and infrequently-run compacting collector. The results show that, in most cases, the programs perform well with this collector; in the remaining cases, they should perform well with a simple and infrequently-run generational compacting collector.

The direct-mapped caches considered in this investigation are limited to certain write policies, and it is assumed that the memory system is capable of handling the write activity of the test programs. The final section of the chapter discusses and justifies these decisions.

2.1. Test programs

The test programs are written in Scheme, a lexically-scoped dialect of Lisp that supports first-class procedures [1, 60]. The primary reason for choosing Scheme was the availability of both a high-quality implementation, namely the Yale T system, and a set of realistic test programs. The T system contains one of the best Scheme compilers currently available [39, 40, 58, 59]. T has been in production use for several years, and has been used by many people to write nontrivial programs.

Measurements of Scheme programs should be relevant to other modern programming languages. Scheme programs are typically written in a mostly-functional style: Data objects are rarely modified after being created, and programmers are encouraged to create and use data structures freely. Scheme does not, however, enforce a particular style or methodology as do, *e.g.*, CLU and ML [43, 45]. The core linguistic constructs of Scheme are similar, if not identical, to those of many garbage-collected languages. Because of Scheme's expressive power, it can efficiently support constructs that have no direct counterpart in the language, *e.g.*, CLU iterators.

The five test programs and their input data are:

ORBIT, the native compiler of the T system, compiling itself;

IMPS, an interactive theorem prover [21], running its internal consistency checks and then proving a simple combinatorial identity;

LP, a reduction engine for a typed λ-calculus [2, 61], typechecking a complex, non-normalizing λ-term and then applying one million β-reduction steps to it;

NBODY, an implementation of Zhao's linear-time three-dimensional N-body simulation algorithm [65, 76], computing the accelerations of 256 point-masses distributed uniformly in a cube and starting at rest; and

GAMBIT, another Scheme compiler [22], quite different from ORBIT, compiling the machine-independent portion of itself.

These programs represent several different kinds of applications and programming styles.

The programs vary in size, but each allocates many megabytes of data and runs for billions of instructions:*

	Lines	Bytes	Insns	Refs	
ORBIT	15,332	94.4M	3.68E9	1.03E9	Scheme compiler
IMPS	42,119	41.1M	4.13E9	1.09E9	Theorem prover
LP	2,981	58.6M	2.21E9	.64E9	λ-calculus reducer
NBODY	857	126.1M	2.43E9	.63E9	N-body simulator
GAMBIT	15,004	106.9M	7.35E9	2.00E9	Scheme compiler

The first column shows the size of each program, measured in lines of Scheme source text. The remaining columns show the number of bytes allocated, the number of instructions executed, and the number of data references made by each program when run, without garbage collection, on its input data. These program runs are significantly longer than those used in previous studies of the cache performance of garbage-collected languages [75, 79].

The programs were compiled and run in version 3.1 of the T system running on a MIPS R3000-based computer [33]. Because all measurements are based upon simulations, the internal details of this machine are unimportant.

The test programs, with their respective inputs, are all non-interactive. The performance of interactive garbage-collected programs depends not only upon program and collector behavior, but upon the cost, in instruction cycles and cache misses, of kernel context switches and user interactions. A study of the cache performance of such programs is beyond the scope of this work.

2.2. Cache design parameters

The portion of the cache design space considered in this investigation is limited in several ways.

Only direct-mapped caches are considered. Because they are the simplest to implement, direct-mapped caches have faster access times than other types of caches [30, 56]; they are therefore the most common type of cache in high-performance computers.

Only one level of caching is considered; no attempt is made to measure the performance of memory systems with multi-level caches. The results reported here are expected to extend to the two- and even three-level caches that are becoming common. An informative analysis of multi-level cache performance, however, re-

*The single letter 'K' denotes a multiple of 2^{10}, the single letter 'M' denotes a multiple of 2^{20}, and the single letter 'G' denotes a multiple of 2^{30}. The single letter 'B' stands for 'byte.' The notation 'aEb' abbreviates '$a \times 10^b$'.

quires a more sophisticated memory-system simulator than that employed here, so a thorough investigation is left to future work.

A wide range of cache sizes is considered, from 32KB to 4MB. This range includes current typical sizes for single-level off-chip caches (32–64KB) and for second- or third-level caches in multi-level systems (1–4MB).

The cache-block size ranges, in powers of two, from 16 to 256 bytes. Most Scheme objects are just a few words long, so, at most sizes, a cache block typically contains many Scheme objects. Main memory will often be discussed in terms of memory blocks, which are assumed to be the same size as cache blocks. The fetch size, *i.e.*, the unit of transfer between the cache and main memory, is also assumed to be equal to the block size.

Only a write-miss policy of write-validate is considered [31]. This policy should perform better than any other for garbage-collected programs; it will be discussed further in §2.5.

Finally, only data-cache performance is studied. Instruction caches are expected to perform reasonably well for Scheme programs, but an investigation of instruction-cache performance is beyond the scope of this work.

2.3. Cache performance without garbage collection

The control experiment measures the cache performance of the five test programs when run without garbage collection. The results will be described in terms of cache overheads, which measure the temporal costs of cache activity relative to the programs' idealized running times.

In a computer system, the time spent by the processor waiting for the memory system is not a function of misses per reference, but of misses per instruction cycle and the number of cycles required to service each miss, which may not be constant. The situation is further complicated by stalls due to other components of the memory system, such as the main memory, write buffers, and virtual-memory translation-lookaside buffers. Thus, to obtain precise figures on the temporal cost of cache misses requires an elaborate simulation of the entire memory hierarchy as well as the relevant parts of the processor pipeline [10, 56].

The cache simulator constructed in the course of this investigation is incapable of such accuracy. In this chapter, cache overheads are calculated under the assumption that the memory system is capable of handling the write activity of the test programs without imposing significant additional costs. Thus cache overheads include only the direct cost of servicing *read misses*, *i.e.*, the cost of fetching memory blocks in response to load instructions; the cost of handling stores is ignored. In §2.5, it will be argued that properties of practical memory systems and of the

programs themselves imply that the write overheads of the test programs should be small.

The time required to service a read miss by fetching the target memory block into the cache, *i.e.*, the *miss penalty*, depends upon details of the main-memory system and upon the block size. In particular, the miss penalty varies directly with the block size, since more time is required to transfer larger blocks in a given memory system. The miss penalties used in the calculation of cache overhead are taken from the high-performance main-memory system studied by Przybylski [56, §3.3.2]. This memory has an address setup time of 30ns, an access time of 180ns, and a transfer time of 30ns for each 16 bytes transferred. Thus a transfer of n bytes requires $30 + 180 + 30 \times \lceil n/16 \rceil$ nanoseconds. The recovery time, *i.e.*, the time required between successive memory transactions, is 120ns; it is ignored in these calculations.

Two hypothetical processors are considered. The slow processor, representing currently-available workstation-class machines, has a cycle time of 30ns (*i.e.*, a 33 megahertz clock); the fast processor, representing high-performance machines available in the near future, has a cycle time of 2ns (a 500 megahertz clock). With these cycle times, the miss penalties for the various block sizes, measured in processor cycles, are:

Block size	16	32	64	128	256	(bytes)
Slow penalty	8	9	11	15	23	(cycles)
Fast penalty	120	135	165	225	345	

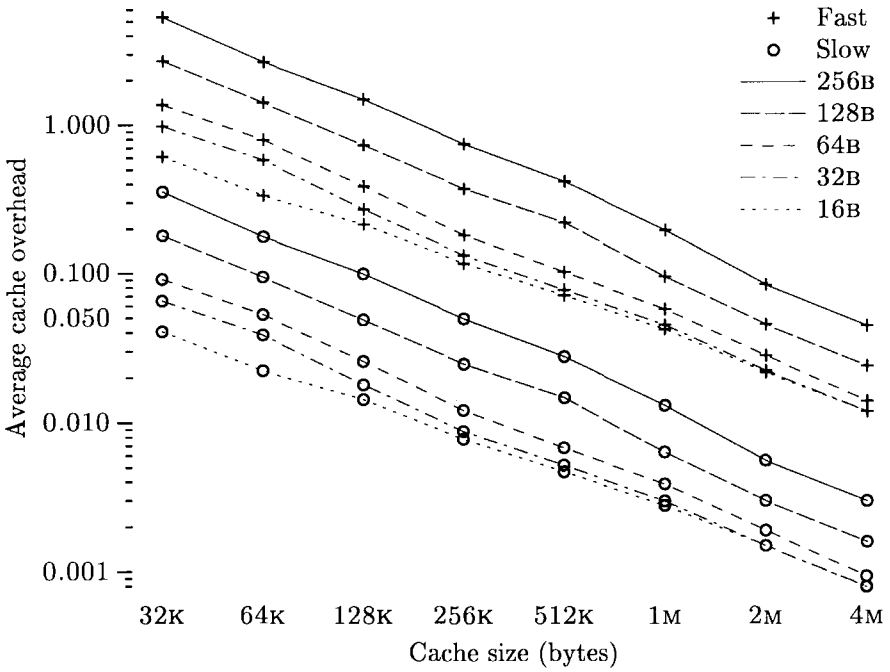
The hit time, *i.e.*, the time required to access a block that is already in the cache, is assumed to be one cycle for both processors. Thus, if a reference hits in the cache, the processor does not stall.

The *cache overhead* of a program is the amount of time spent waiting for read misses expressed as a fraction of the program's idealized running time, in which no misses occur and one instruction completes in every cycle. That is,

$$O_{cache} = \frac{M_{prog} \times P}{I_{prog}} ,$$

where M_{prog} is the total number of read misses during the program run, P is the miss penalty, in processor cycles, and I_{prog} is the total number of instructions executed by the program. The more familiar metric of *cycles per instruction* [10] is one plus the overhead of each cache in the memory hierarchy.

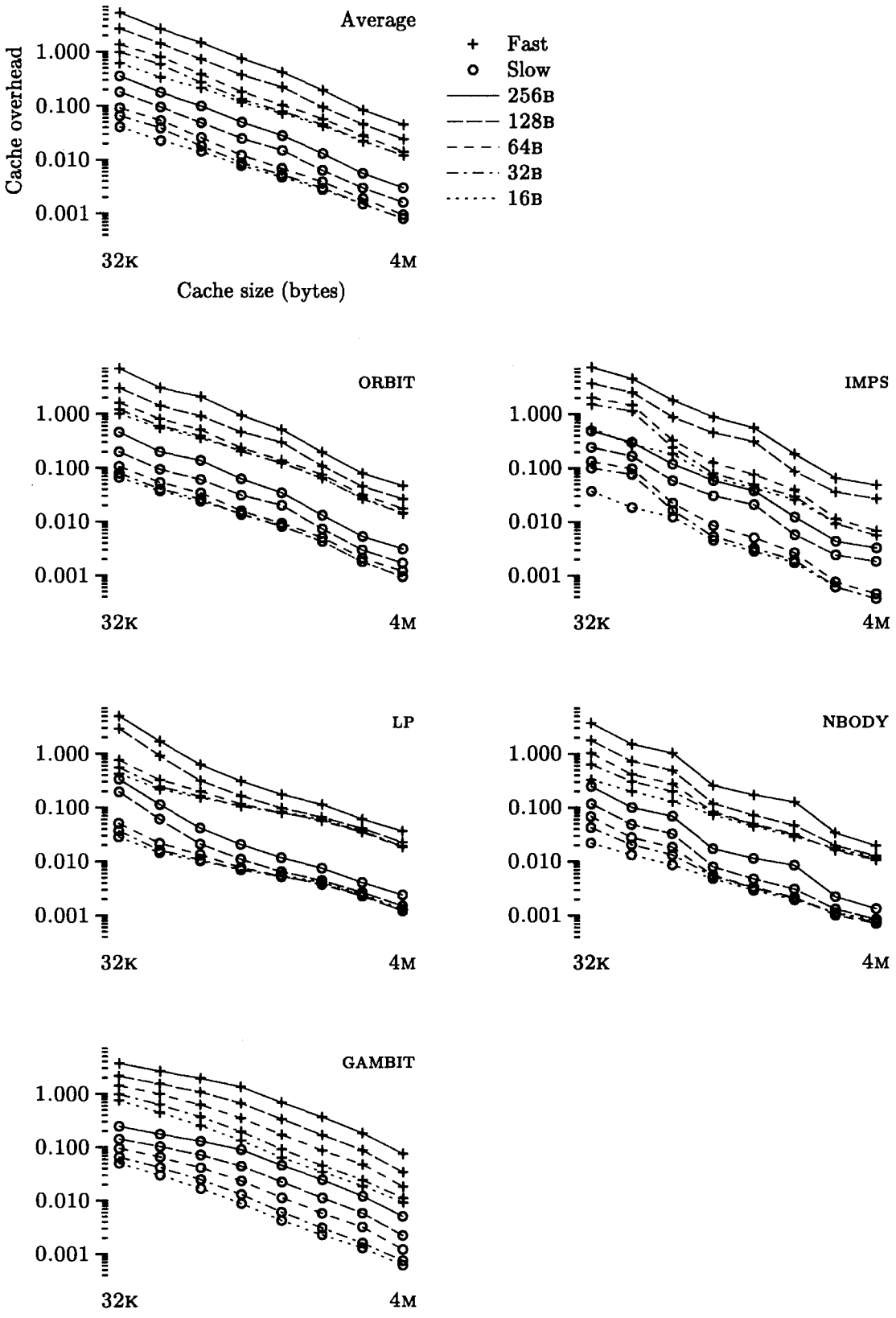
With these assumptions and definitions in hand, the cache overhead for the test programs, run without garbage collection, can be calculated:



There are two sets of curves in this graph, one for each of the hypothetical processors. The height of a data point shows the average cache overhead, across all programs, for the given block size, cache size, and processor speed. (Separate cache overheads for each program are shown on the following page.)

For the slow processor, even a small 32KB cache has a cache overhead of less than five percent when the block size is 16 bytes. For the fast processor, a 1MB cache is required in order to achieve a similar overhead, but fast machines are expected to have caches at least that large. Caches in such machines may employ larger block sizes, but, with a sufficiently large cache, it is still possible to achieve an overhead of less than five percent. For both processors, smaller block sizes always yield superior performance.

The control experiment has revealed, then, that when run without any garbage collection at all, the test programs have excellent cache performance. There is so little room for improvement—five percent or less—that it is unclear whether improving cache performance should even be a priority. More significant improvements in overall program performance might be attained by improving other aspects of the hardware, the language system, or the programs themselves. No method for improving cache performance that imposes a significant overhead of its own could be effective.


Average
Cache overhead
1.000
0.100
0.010
0.001
32K
4M
Cache size (bytes)
+ Fast
o Slow
256B
128B
64B
32B
16B
ORBIT
IMPS
LP
NBODY
GAMBIT

2.4. Program performance with a simple collector

The second experiment measures the cost of running the test programs with a modest amount of memory and a simple, efficient, and infrequently-run compacting collector. This collector performs well in most cases; a simple generational compacting collector should perform well in the remaining cases.

Garbage-collection overhead. During a program run, a garbage collector imposes both direct and indirect costs. Directly, the collector itself executes I_{gc} instructions and causes M_{gc} cache read misses. The magnitude of I_{gc} depends upon the amount of work done by the collector; that of M_{gc} depends upon the collector's own memory reference patterns.

Indirectly, there are two ways in which the collector affects the number of misses that occur while the program is running. Each time the collector is invoked, its memory references remove some, or possibly all, of the program's state from the cache; when the program resumes, more cache misses occur as that state is restored. The collector can also move data objects in memory, which may improve (or degrade) the objects' reference locality, thereby decreasing (or increasing) the program's miss count. These effects are together reflected in ΔM_{prog}, which is the change in the program's miss count relative to M_{prog}, its miss count when run in the same cache without garbage collection. If the collector improves the program's cache performance by more than enough to make up for the cost of restoring the program's cache state after each collection, then ΔM_{prog} will be negative.

The collector can also cause the program to execute ΔI_{prog} more instructions. This occurs in the T system because hash-table keys are computed from object addresses. Because the collector can move objects, each table is automatically rehashed, upon its next reference, after a collection. The cost of rehashing, in instructions and in cache misses, is usually small.

When run with a given collector, the *garbage-collection overhead* of a program is the sum of these temporal costs, expressed as a fraction of the program's idealized running time. That is,

$$O_{gc} = \frac{(M_{gc} + \Delta M_{prog}) \times P + I_{gc} + \Delta I_{prog}}{I_{prog}},$$

where P is again the miss penalty, in processor cycles, and I_{prog} is the total number of instructions executed by the program. Because ΔM_{prog} can be negative, it is possible for O_{gc} to be zero or negative, which will be the case if the collector improves the program's cache performance by more than enough to pay for its own running cost.

The combined cache and garbage-collector overhead is simply $O_{cache} + O_{gc}$. Because O_{gc} can be negative, it is possible for the combined overhead to be zero, although this would require an impossibly perfect collector that executes no instructions, causes no cache misses, eliminates all of the program's cache misses, and does not cause the program to execute any extra instructions. The goal of methods such as aggressive collection is to achieve an overhead that is sufficiently negative to counter O_{cache} significantly.

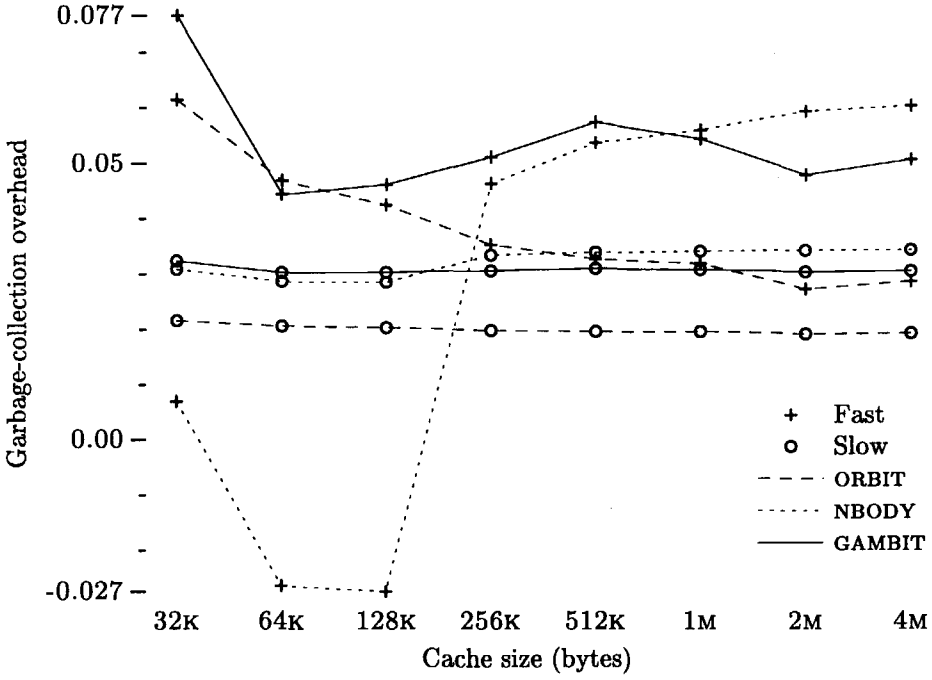
A simple collector. The garbage collector used in this experiment is a straightforward implementation of Cheney's algorithm for copying compacting collection, which is perhaps the simplest practical collection technique known [14].

Cheney's method uses two memory areas of equal size, sometimes called *semispaces.* At any given time, one semispace acts as the *new* area, while the other is the *old* area. The heap is contained entirely within the new area, and objects are allocated linearly in the new area from a contiguous run of free memory. When the free memory is exhausted, a collection is performed. The collector first exchanges, or *flips*, the roles of the areas; it then copies all live (*i.e.*, non-garbage) objects from the old area into a contiguous portion of the new area in a breadth-first manner. Cheney's algorithm avoids the need for a recursion stack during the copying process by using pointers into the new area to distinguish between copied objects that can still point to uncopied objects from those that cannot.

Although the collector requires two areas, it is not necessary that both always fit within the available physical memory. Between collections, only one area need be resident. The old area is accessed randomly during a collection, but the new area is accessed linearly as live objects are copied into it and scanned. Thus, for reasonable performance during a collection, enough physical memory is required to hold the old area, a portion of the new area, and whatever working space is required by the collector. As long as the virtual-memory system can handle the temporary increase in memory demand, few page faults should occur. What faults remain can be reduced further by arranging for the collector to advise the virtual-memory system of its expected usage patterns [17].

Like the definition of O_{cache}, the definition of O_{gc} above does not include the cost of handling memory writes. The Cheney collector allocates and initializes memory in a linear fashion, just as a program does. Therefore the argument that write overheads should be low, in §2.5, applies to the collector itself as well as to the test programs.

Program performance. When run with the Cheney collector, configured to use 16MB semispaces, the garbage-collection overheads (O_{gc}) of three of the five test programs are fairly low:



The overheads for IMPS and LP, discussed below, are not shown here.

This graph shows data for 64-byte blocks; overheads for other block sizes are similar. There is one set of curves for each of the hypothetical processors; in each set, there is one curve for each of three of the test programs. The height of a data point shows the measured collection overhead for a program when run with the Cheney collector in a cache of the indicated size. With the slow processor, all overheads are less than four percent; with the fast processor, overheads are usually higher, reaching a maximum of 7.7%, but are still acceptable.

For each program, the variations in collection overheads are due to the number of cache misses caused by the collector itself and to the collector's effect upon the program's miss count. Even this simple collector might improve a program's cache performance by compacting live data objects in memory, so another source of variation is the extent to which this type of improvement occurs. For a given cache size and processor speed, the cache overheads differ because of these factors and because the amount of work done by the collector is program-dependent, being a function of the number of live objects at the time of each collection.

For one program, garbage-collection overhead is negative in two cases, indicating a significant improvement. These negative overheads are not, however, due to

a general improvement of the program's reference locality by the collector. When run without collection, the program in question, NBODY, has a few memory blocks that thrash in sufficiently small caches. That is, the memory blocks map to the same cache block, and they are referenced in such a way that they cause many misses. The Cheney collector happens to move the objects involved, thereby eliminating the thrashing behavior and significantly reducing the number of misses. This improvement is not as noticeable in a 32KB cache because there are so many more misses in a cache of that size to begin with; it does not occur in caches larger than 128KB because these memory blocks map to different cache blocks in larger caches. In §4.1, methods for eliminating thrashing that do not involve a garbage collector will be presented.

The previous graph only shows garbage-collection overhead data for ORBIT, NBODY, and GAMBIT. IMPS suffers from a more extreme case of the thrashing behavior just described, so its overheads are highly variable. When thrashing does not occur, the overheads for IMPS are comparable to those shown above.

Overheads for LP are not shown because they are uniformly 40% or higher. LP creates a large data structure that grows monotonically in size until the end of its run. Thus, unlike the other programs, the amount of work done by the Cheney collector in successive collections increases, since it must copy this structure each time. A simple generational collector would avoid this problem [4, 42]; although it would impose costs beyond those of the Cheney collector, the work avoided by not repeatedly copying long-lived structures should more than counter those costs. Like the Cheney collector, a generational collector should be run infrequently in order to take advantage of the programs' naturally good cache performance.

The collection overhead of an aggressive garbage collector is likely to be significantly higher than that of an infrequently-run generational collector. As proposed by Wilson *et al.* and by Zorn, an aggressive collector is essentially a generational collector with a new-object area, or first generation, that is sufficiently small to fit mostly or entirely in the cache [75, 79]. An aggressive collector will thus incur all the costs of an ordinary generational collector, including the overheads of managing several generations and of detecting and updating pointers from old objects to new objects. An aggressive collector will spend more time copying objects from the new-object area, for more frequent collections leave less time for new objects to become garbage before being copied to the next generation. It seems likely that this added copying cost will be significantly larger than the meager improvement in cache performance that is possible. Thus, even if an aggressive collector could reduce cache overhead to zero, it would be unlikely to pay for its cost over that of an infrequently-run generational collector.

2.5. Cache write policies and write overhead

In this investigation, only caches with a write-miss policy of write-validate are considered. The first part of this section reviews the possible write-miss policies and argues that write-validate policies, or some equivalent mechanisms, are likely to become common.

In this chapter, for the purpose of calculating cache overhead, it was assumed that the memory system is capable of handling the write activity of the test programs without imposing significant temporal costs. The justification of these decisions requires understanding write-hit policies and the relationship between write activity and available memory bandwidth. The second part of this section reviews the possible write-hit policies and argues that the write overheads of the test programs should be small.

Write miss policies. A *write miss* occurs when the target of a store instruction is a memory block that is not in the cache.* When designing a cache, the first choice to be made in deciding how to handle write misses is whether or not to fetch the target memory block into the cache. If a *fetch-on-write* policy is used, then a write miss will stall the processor while the target block is fetched into the corresponding cache block. If it is decided that a write miss will not trigger a fetch, then there is another choice to be made, namely whether or not to allocate the cache block to the target memory block. If the cache block is allocated, then the resulting policy is called *write-validate.* If the cache block is not allocated, the resulting policy is called either *write-around* or *write-invalidate*, depending upon whether or not data is written to the cache in parallel with the tag check; with both policies, the stored data is sent directly to main memory, so a future load of that data will cause a read miss and an ensuing fetch.

A write-validate policy requires that a validity bit be associated with each word in a cache block. When a write miss occurs, the cache block is allocated to the target memory block by setting the tag bits, but the contents of the memory block are not fetched into the cache.** The stored data is written into the cache block and the validity bits associated with that data are set; all other validity bits in the cache block are cleared. If every word in the memory block is written before it is read, then the original contents of the block will never be fetched.

Jouppi has demonstrated that, for programs written in conventional languages, a write-validate policy always outperforms fetch-on-write and write-invalidate, and usually outperforms write-around [31]. In particular, for his test programs a write-

*The following discussion of write policies is based in part upon a recent paper by Jouppi [31].

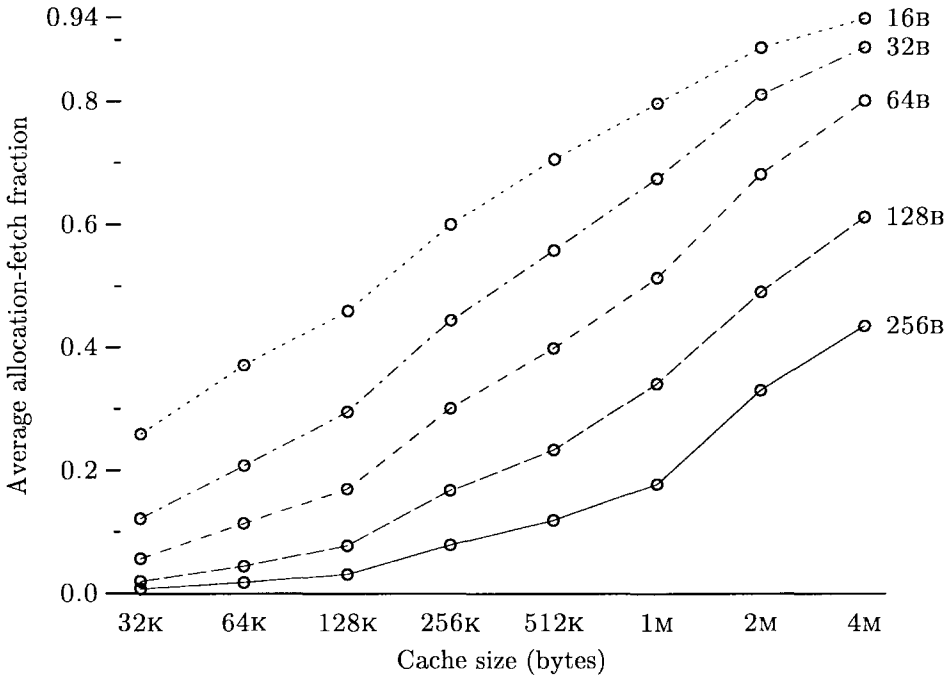
**If the write-hit policy is write-back, then the prior contents of the cache block are flushed to memory at this time.

validate policy eliminates, on average, roughly one-third of the cache misses seen with fetch-on-write. Write-validate works well because it avoids useless fetches, and because recently-written data is usually referenced sooner than the prior contents of the cache block containing that data.

For programs written in garbage-collected languages, a write-validate policy should be superior for the same reasons. Because of allocation activity, useless fetches can be a significant problem for such programs. This is especially true when objects are allocated linearly from a contiguous run of free memory, which is the case when no garbage collection is done or when a compacting collector is used. When an object is allocated, its component words are initialized in ascending order of address. If the object contains the first word of a memory block, then a type of write miss called an *allocation miss* occurs when that word is initialized. With a fetch-on-write policy, the miss will cause the contents of the memory block to be fetched. The information retrieved by this *allocation fetch* will never be used, however, because every word in heap memory is initialized before it is read.

The other three write policies avoid allocation fetches, but write-around and write-invalidate are incapable of taking advantage of the high temporal and spatial locality of object references. It will be seen in Chapter 3 that most heap-memory blocks are not referenced after their associated cache blocks are required for newer memory blocks. Thus it is more likely that a program will read a recently-allocated memory block than an older block that happens to map to the same cache block. With a write-around or write-invalidate policy, the first load from any heap-memory block will cause a read miss; with a write-validate policy, the first load from a recently-allocated block is likely to hit in the cache. Since references to recently-allocated blocks are expected to be more common, write-validate should outperform both write-around and write-invalidate.

For the Scheme test programs, a write-validate policy easily outperforms a fetch-on-write policy. If a fetch-on-write policy is used, then, when the test programs are run without garbage collection, allocation fetches become a significant fraction of all fetches as the cache size increases and as the block size decreases:



This graph contains one curve for each block size, and one data point for each combination of cache and block sizes. For a given cache size, the average allocation-fetch fraction increases as the block size decreases because the number of allocation misses depends inversely upon the block size; *e.g.*, if the block size is halved, there will be (approximately) twice as many allocation misses. In larger caches, allocation fetches can easily account for half or more of all fetches.*

An alternative way to avoid most useless fetches is to use a *cache-block-allocation* instruction, first described by Radin [57] and apparently reinvented, in the context of garbage-collected languages, by Peng and Sohi [53]. Given a memory block address, this instruction allocates the associated cache block but does not fetch the contents of the memory block. Such an instruction has appeared in a number of machines [19, 32, 48], but it is not an ideal solution. A cache-block allocation instruction can be costly to use correctly, since all old data in the cache block must be known to be useless. For a garbage-collected language, this implies that the instruction can only be invoked when the first word of a block is allocated;

*Due to limitations of the cache simulator, this graph underestimates, probably only slightly, the actual fraction of allocation fetches.

with the cache sizes being considered, but an overhead of less than ten percent is attainable with a 2MB cache.

With a write-validate policy, the cache overhead for a given cache size varies inversely with the block size for the block sizes under consideration. In contrast, the above graph shows that, with a fetch-on-write policy, larger blocks have an increasing advantage as the cache size increases. This is because smaller blocks entail more allocation fetches and because, in larger caches, more misses are allocation misses, regardless of the block size.

Write hit policies and write overhead. A *write hit* occurs when the target of a store instruction is a memory block that is already in the cache. There are two ways to handle write hits. With a *write-through* policy, the stored data is written both to the cache block and to main memory. With a *write-back* policy, the stored data is written only to the cache block; the contents of the cache block are written back to memory later, when the cache block is allocated to a different memory block. Each policy has its own advantages. A write-through policy is simpler to implement, can provide higher bandwidth into the cache, and has other advantages important to on-chip caches [31]. A write-back policy, in contrast, can exploit the reference locality of stores in order to reduce write traffic to main memory.

Whatever the write-hit policy, if memory-write transactions occur too frequently, the processor will stall while waiting for them to complete. The remainder of this section argues that because stores in the test programs have high temporal and spatial locality, the overhead of the write traffic generated by a write-back cache should be small, even when the Cheney collector is used.

Among the memory areas in the T system, most stores are to the procedure-call stack:

	ORBIT	IMPS	LP	NBODY	GAMBIT
Stack	71.36	90.24	71.05	59.34	87.54 %
Static	2.75	1.49	4.47	15.55	1.63
Heap	25.87	8.26	24.47	25.09	10.81

It will be seen in Chapter 3 that the stack is a highly local structure, with most stack references occurring in a small contiguous group of memory blocks. Most references to static memory are also confined to a small, though noncontiguous, set of blocks, so most static stores are likely to be similarly concentrated. A write-back policy will therefore be best for stack and static references since it will not require a memory transaction every time a frequently-referenced memory block is modified. Because such blocks tend to reside in the cache, they will rarely be written back to memory.

Stores to the heap are uniformly spread through heap memory. In the mostly-functional programming style typically used in Scheme programs, data objects are usually written only once, when they are initialized.* As a consequence, most heap stores are initialization stores to newly-allocated memory blocks; non-initialization stores to older heap blocks are rare. This is evident when the above heap-store percentages are separated into initialization and non-initialization stores:

	ORBIT	IMPS	LP	NBODY	GAMBIT
Init	20.93	7.56	22.55	25.08	9.86 %
Non-init	4.94	0.69	1.91	0.01	0.94

Objects are allocated linearly in memory, so all initialization stores to a given memory block occur close together in time. The Cheney collector copies objects to the new area in essentially the same manner, so the store behavior of the programs is not much different when they are run with this collector. A write-back cache will coalesce initialization stores so that each block is likely to be written to memory only once, some time after its last initialization store. The same effect could be achieved by augmenting a write-through cache with either a coalescing write buffer or an auxiliary write cache [31]. Such an arrangement, however, would probably not be as effective at reducing the write traffic due to stack and static stores, which tend to be concentrated in a small number of different blocks.

With a write-back cache, initialization stores should not generate excessive write traffic. Once the first word of a memory block has been initialized, the block will be flushed from the cache and written to main memory only when its cache block is allocated to some other memory block. Chapter 3 will show that most cache blocks see little interference in the test programs, which suggests that most newly-allocated memory blocks will only be flushed when their cache blocks are next allocated for an even newer block. Thus the rate at which heap blocks are written back to memory is likely to be closely related to the object-allocation rate.

The cache simulator is incapable of measuring the relationship between the heap-block write-back rate and the allocation rate. The write overhead due to initialization stores in an improbable near-worst case, however, can be estimated. Suppose that a program repeatedly allocates exactly one block of data and then immediately references some other block that shares the same cache block, thereby flushing the newly-allocated block. Let B be the block size, in words, and let D be the minimum delay, in cycles, between successive writes. At least one instruction is required to initialize each word in a block, so once the write buffer, if any, is full, the processor will repeatedly spend B cycles initializing a new block and at most

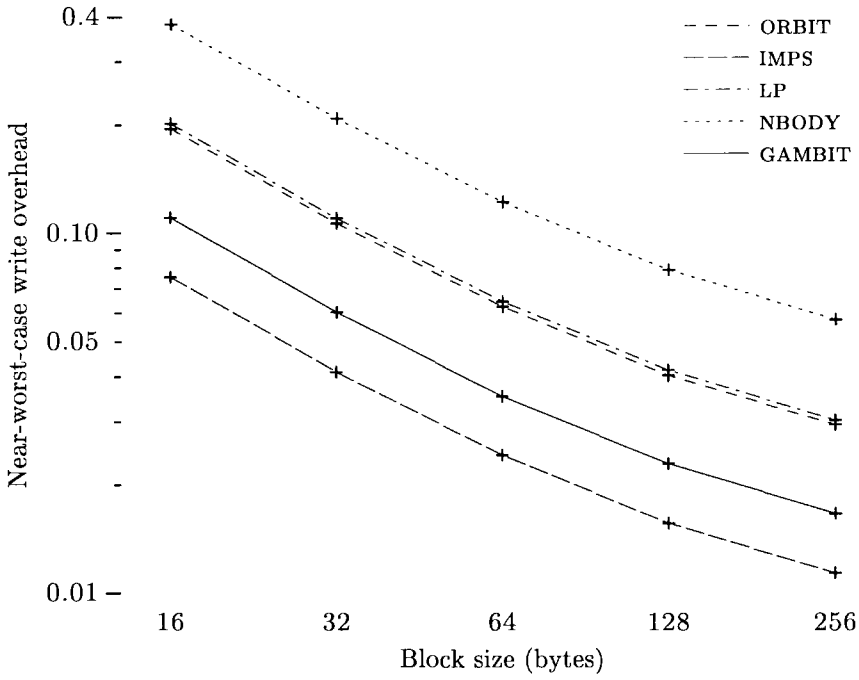
*Actually, the T runtime system initializes most objects twice, in rapid succession.

$D - B$ cycles waiting for a write transaction to complete. The write overhead of this hypothetical reference pattern is

$$O_{istore} \leq \frac{W \times (D - B)}{I_{prog}} ,$$

where W is the total number of words allocated and I_{prog}, as before, is the total number of instructions executed by the program.

Assume that, for a given processor speed and block size, the write latency D is equal to the miss-penalty time shown on p. 8.* Then for the test programs, the write overhead of this reference pattern for the hypothetical slow processor is zero for blocks sizes of 64 bytes or larger, since more time is spent initializing each block than is required between successive writes; for 16- and 32-byte blocks, D is at most twice B, and O_{istore} is well under one percent. For the fast processor, O_{istore} is usually less than ten percent:



This graph contains one curve for each program; each data point shows the value of O_{istore} for the indicated block size. The estimated write overheads for this near-worst case are small; because memory blocks are actually allocated in a much more sedate manner, the actual write overheads due to initialization stores should be at most a few percent.

*In fact, the write latency is usually smaller [56, p. 30].

2.6. Conclusion

This chapter has established two key facts about the Scheme test programs.

First, they have excellent cache performance when run without any garbage collection at all; improving cache performance hardly seems necessary. Furthermore, any method for improving cache performance that imposes a significant overhead of its own will not be effective; such a method will not be able to recover its own running cost, let alone improve the overall performance of the client program.

Second, for these programs, the overhead of a simple, infrequently-run compacting collector is acceptably low in most cases. An infrequently-run generational compacting collector should yield good performance in the remaining cases.

These results apply only to the five test programs. In order to generalize these results to other programs, and to other programming languages, the next chapter analyzes the memory behavior of the programs in order to understand why their cache performance is so good.

3. Analysis of memory behavior

The first experiment reported in Chapter 2 established that the test programs have excellent cache performance when run without a garbage collector or any other mechanism that might improve reference locality. This chapter explains that observation by analyzing the memory behaviors of the test programs in the same context. The analysis will establish that, when run without garbage collection, the test programs have good cache performance because their memory behaviors are naturally well-suited to direct-mapped caches. In Chapter 4, the measurements and conclusions of the analysis will be used to identify three methods by which the test programs' performance might be improved, and the analysis will be generalized to other programs and other programming languages.

The conclusions of the analysis are not limited to the idealized setting of a program running without a garbage collector. Chapter 2 argued that the test programs will perform well with infrequently-run compacting collectors, which can take advantage of the programs' naturally good cache performance. The memory behavior of a program running with such a collector will differ from the idealized case in two ways. First, it will be interrupted, albeit rarely, by collector invocations. Second, the collector may move objects in memory; as was argued in Chapter 2, these actions will have significant effects only if they happen to eliminate thrashing. Aside from these differences, the program's memory behavior should be similar to that of the idealized case. In particular, in the long intervals between collections, objects will be allocated linearly from a large free-memory area, approximating the idealized case of linear allocation from an unbounded area.

Some foundations must be laid before proceeding with the analysis. After defining the notion of memory behavior, the memory layout of the Scheme system being studied will be briefly described. A plot of the cache misses that occur during part of a program run will be examined in order to develop a visual idea of the connection between memory behavior and cache activity. Finally, a coarse-grained unit of time and a notion of interference in the cache between memory blocks will be defined.

The analysis will then show that most memory blocks in the test programs have very short lifetimes and are not referenced many times. These facts, together with the object allocator's linear sweep through memory, imply that most memory blocks are spread through time and space in such a way that references to them cause little cache interference. Nearly all other blocks are not very active and are also not referenced many times, so they too cannot cause significant interference. Long-lived and frequently-referenced blocks, called *busy* blocks, are very rare, and turn out to improve cache performance more often than they degrade it.

3.1. Memory behaviors, memory blocks, and memory areas

The *memory behavior* of a program run is a record of the program's computational actions with respect to main memory. Memory behaviors include allocation actions as well as read and write actions, and therefore contain more information than simple address traces. In general, a memory behavior also includes instruction references; these are unnecessary here, however, since the analysis only seeks to explain data-cache performance.

The analysis of memory behaviors is carried out in terms of fixed-size memory blocks. Scheme objects might seem a more natural unit, but blocks, not objects, are the fundamental unit of transfer in practical memory systems. For simplicity, the cache- and memory-block size is fixed at 64 bytes; other block sizes will be discussed at the end of the chapter. Most Scheme objects are comparatively small, so a 64-byte block typically contains a handful of objects.

Memory in the Yale T system is divided into four contiguous *areas*, which the analysis will consider in turn.*

The *dynamic* area contains data objects created by programs as they run. The *allocation pointer* contains the address of the next available word in the dynamic area, and is incremented by each allocation action. When a program is run without collection, the allocation pointer starts at the base of the dynamic area and grows upward, without bound, until the end of the run. When a compacting collector is used, the allocation pointer starts at the base of the new-object area and grows upward until the end of the area is reached, at which time the collector is invoked.

The *static* and *loaded-data* areas are similar, and will sometimes together be referred to as the *static/loaded* areas. The static area contains data structures and code for the compiler, library, and runtime system, and is identical for all program runs. The loaded-data area contains data and code loaded from files, or created, before the program starts running; it thus typically contains the program itself.** Because instruction references are ignored, code objects are only referenced as they are loaded or created.

Finally, the *stack* area contains the procedure call stack. Call frames are contiguous and usually quite small, on the order of several words each.

A block is considered *active* if it is referenced at least once. Active blocks are distinguished from inactive blocks because programs typically touch only a small fraction of the compiler, library, and runtime-system data in the static area. While all blocks in other areas are active, most static blocks are inactive.

*Strictly speaking, this is true only when the garbage collector is disabled; when the collector is enabled, dynamic and loaded objects are spread among several areas.

**All program measurements begin at the first instruction executed by the T system, so they include the loading process.

In the test programs, most active memory blocks are dynamic:

	ORBIT	IMPS	LP	NBODY	GAMBIT
Dynamic	99.39	70.54	98.75	99.73	98.13 %
Static	0.45	0.74	0.44	0.11	0.26
Loaded	0.14	28.71	0.79	0.15	1.54
Stack	0.02	0.01	0.02	0.00	0.08

ORBIT has few active loaded blocks because it is the T system's compiler, and therefore resides in the static area. The loaded-block percentage for IMPS is large because IMPS is a large system; that for NBODY, in contrast, is small because the program itself is small but it allocates much memory, more than any of the other programs.

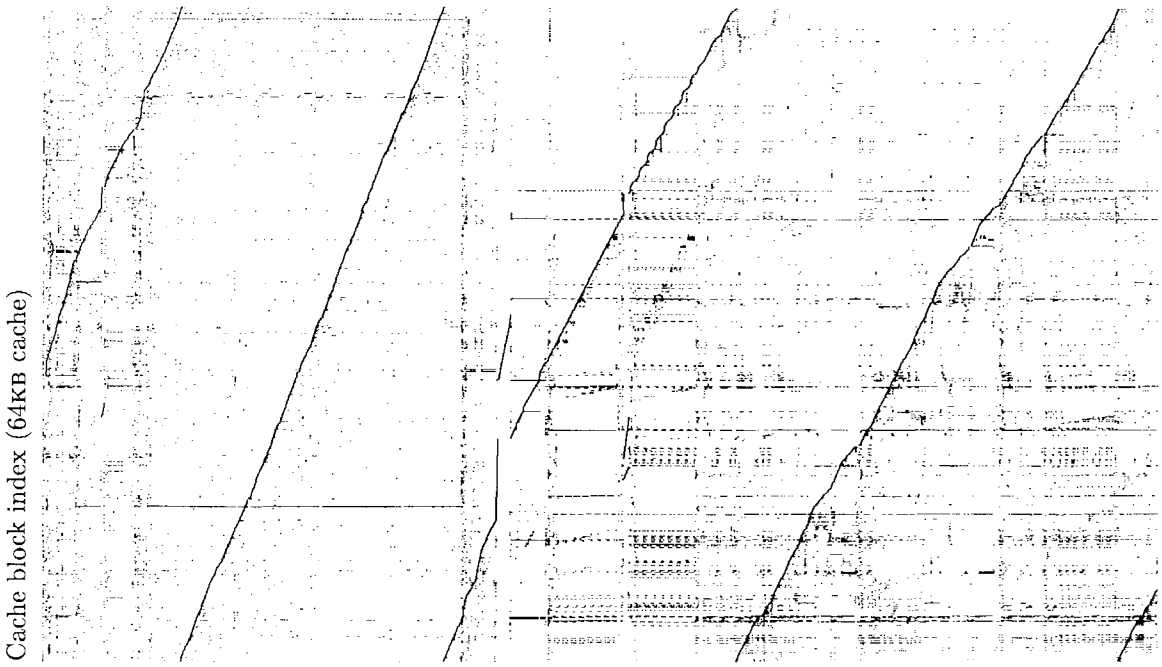
While most active memory blocks are dynamic, most references are to non-dynamic blocks:

	ORBIT	IMPS	LP	NBODY	GAMBIT
Dynamic	20.27	11.64	13.24	16.60	12.80 %
Static	41.70	24.46	16.25	43.03	29.02
Loaded	0.25	25.81	40.27	3.95	11.95
Stack	37.73	37.95	30.23	34.49	46.22

There is no obvious connection between the number of references in a non-dynamic area and the number of blocks in that area. In particular, the stack area in each program accounts for at least 30% of all references, yet it contains hardly any blocks.

3.2. Cache miss patterns

When cache misses are plotted as a function of time, various patterns become apparent:



The horizontal x axis of this plot is calibrated in data references, which are the fundamental time unit of the analysis; there are 1024 references for each dot width. On the vertical y axis, there is one dot width for each of the 1024 blocks of a 64KB, 64B-block direct-mapped cache. A dot is shown at (x, y) if at least one miss occurred in cache block y during the x^{th} 1024-reference interval. This plot shows cache misses for the first 1,784,831 references of a short run of ORBIT.

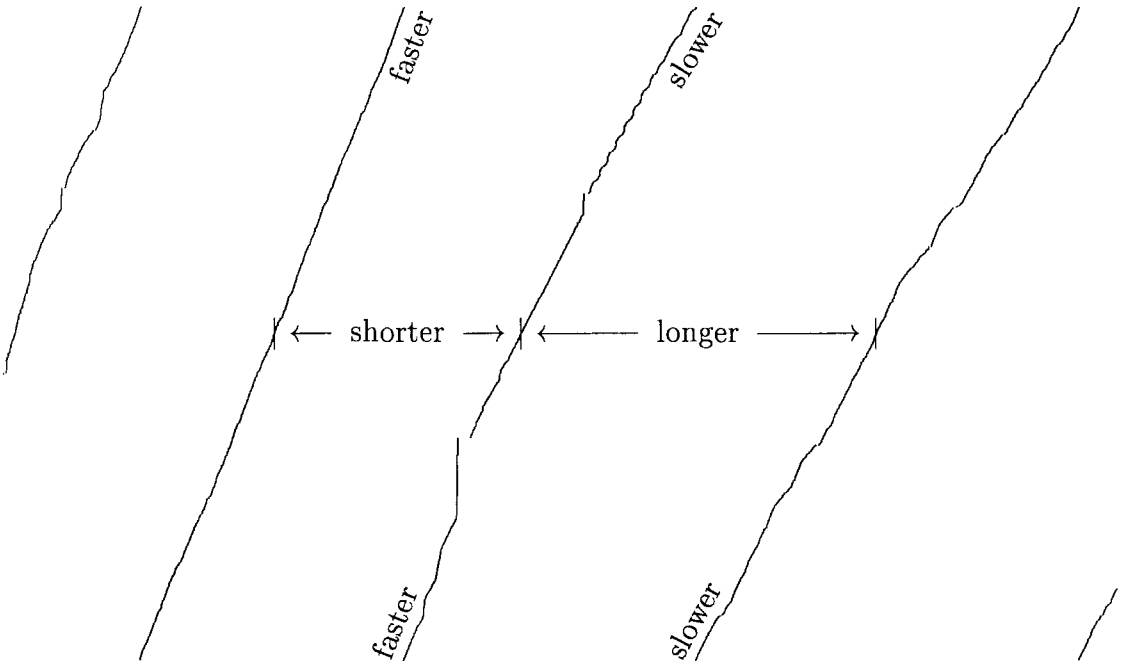
The most prominent pattern is that of the diagonal stripes, which are due to allocation misses. When an object is allocated, the allocation pointer is first incremented by the new object's size; then the component words of the new object are initialized in ascending address order. Each time one of these initialization stores reaches a new dynamic memory block, an allocation miss occurs, flushing the cache block and allocating it to the newly-allocated block. A direct-mapped cache maps a memory block to a cache block by taking the memory block's index modulo the cache size (in blocks), so the allocation pointer continually sweeps the cache from one end to the other, leaving a trail of allocation misses.

The slope of an allocation-miss stripe at a given point in time reflects the program's allocation rate, relative to its reference rate, at that time. The partial run shown in the plot is typical in that the allocation rate usually changes slowly,

but sometimes changes quickly. For example, the nearly vertical segment in the lower part of the second full stripe indicates that the allocation rate is very rapid for a short time; this is probably due to the allocation of one or a few large objects.

For each cache block, time is divided by its allocation misses into a sequence of *allocation cycles.* An allocation cycle begins at each allocation miss in a block, and ends just before the next allocation miss in that block.*

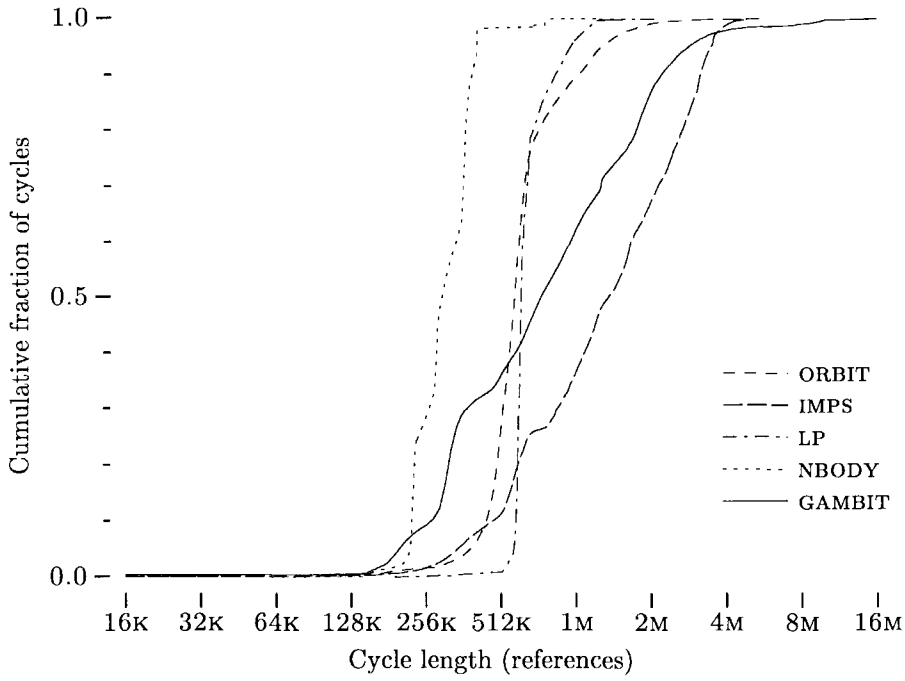
The length of an allocation cycle depends inversely upon the prevailing allocation rate between its defining allocation misses. When the allocation rate is faster, cycles are shorter; when the rate is slower, cycles are longer:



The length of an allocation cycle depends directly upon the size of the cache, as does the total number of cycles that occur in a cache block during a program run. When the cache size is halved, the allocation pointer has half as many blocks to traverse before it revisits a cache block. Thus each allocation cycle is split into two shorter cycles, and the cycle count of each cache block is (approximately) doubled. Conversely, when the cache size is doubled, pairs of adjacent cycles are combined into longer cycles, and the cycle count of each cache block is (approximately) halved.

*For simplicity, the partial cycles at the beginning and end of a program run are ignored.

Allocation cycles in 64KB caches are typically several hundred thousand to two million references in length:



This plot shows, for each test program, the cumulative frequency distribution of the lengths of its allocation cycles in a 64KB, 64B-block cache. A point on one of the curves indicates, in its y value, the fraction of allocation cycles with lengths no greater than its x value. Since each test program runs for at least one-half billion references, most allocation cycles are quite short, around a thousandth or less of the total program running time. Even the longest allocation cycles, which are few in number, account for less than a hundredth of the total running time.

That allocation cycles in a 64KB cache are small relative to total program running times makes them a useful coarse-grained unit of time. If the activity of a memory block is confined to a few 64KB-cache cycles, then its activity is confined to a few short intervals of time. This is true in 64KB caches and in all larger caches, since each allocation cycle in a larger cache contains a power-of-two sequence of 64KB-cache cycles.

3.3. Interference

Intuitively, cache misses occur when memory blocks interfere with each other as they compete to use the same cache block. That several memory blocks map to one cache block, or *collide*, does not imply any interference, let alone significant interference. For example, if the references to each memory block occur in large, non-overlapping groups, then there will be little interference, and therefore few misses, in the shared cache block. On the other hand, if the memory blocks are referenced in a round-robin fashion then there will be maximal interference, *i.e.*, the blocks will thrash, since every reference will cause a miss.

To establish a connection from memory behavior to cache performance, a precise definition of interference is required:

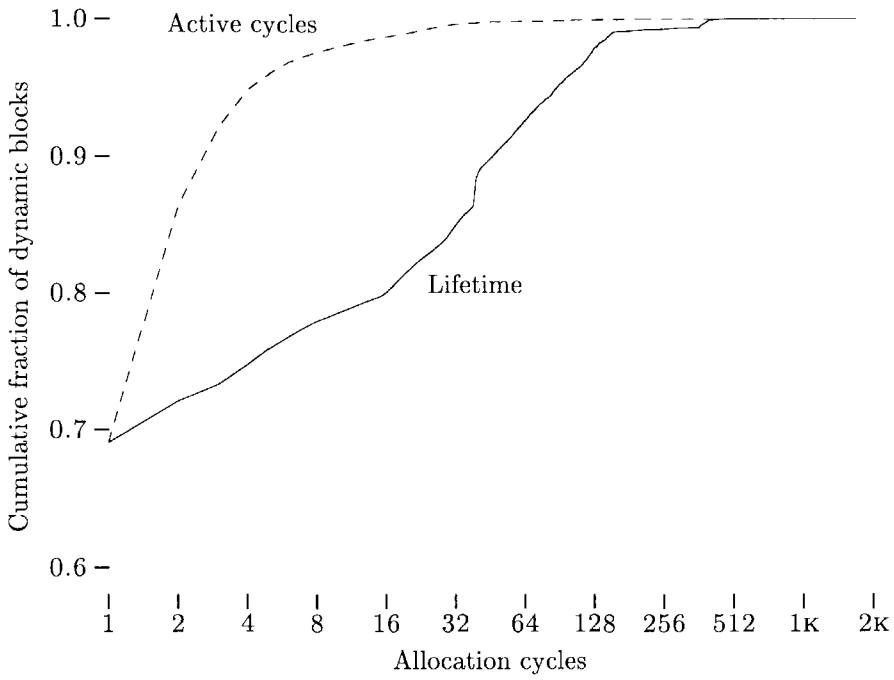
> Memory block a is said to *interfere* with block b if there exists a reference to block a that can cause block b to be removed from the cache before the final reference to b occurs.

Interference is an asymmetric relationship: It is possible for a to interfere with b, but for b never to interfere with a. This simplifies the analysis, for it allows memory blocks to be studied alone, or in classes, as sources of interference. If interference were a symmetric relationship, then the analysis would be complicated by having to consider pairwise combinations of memory blocks or classes of blocks.

If a memory block is removed from the cache before its final reference, a miss will occur the next time it is referenced. This type of miss is called a *restoration miss*, so as to be distinguished from the allocation misses that occur when newly-allocated memory blocks are initialized. Unlike allocation misses, restoration misses always trigger memory fetches.

The definition of interference captures the fact that, relative to the goal of eliminating restoration misses, interference from one block is just as bad as interference from many blocks. For example, suppose that block b is referenced at times t_1 and t_2. If a block colliding with b is referenced between t_1 and t_2, then b will be removed and a restoration miss for it will occur at t_2. If multiple colliding blocks are referenced during this interval, then still only one restoration miss will occur at t_2. To eliminate the restoration miss for b requires eliminating all references to other blocks during the interval; if a reference to just one other block remains, b will still be removed from the cache. Therefore the definition of interference is carefully worded so that if a interferes with b, it is not guaranteed that b will actually be removed by a reference to a; it is guaranteed, however, that b will be removed by a reference to a if no reference to another block does so. To eliminate all restoration misses for b requires eliminating all interference with b.

The extent to which multi-cycle blocks can interfere with one-cycle blocks and with blocks in other areas is limited by the fact that nearly all multi-cycle blocks are only active in a few 64KB-cache allocation cycles. The relative inactivity of multi-cycle blocks can be seen by comparing, for the dynamic blocks in each program, the cumulative frequency distribution of their lifetimes with that of their active-cycle counts, where the active-cycle count of a block is just the number of allocation cycles in which it is active. The most striking example is GAMBIT, which has more long-lived blocks than the other programs:

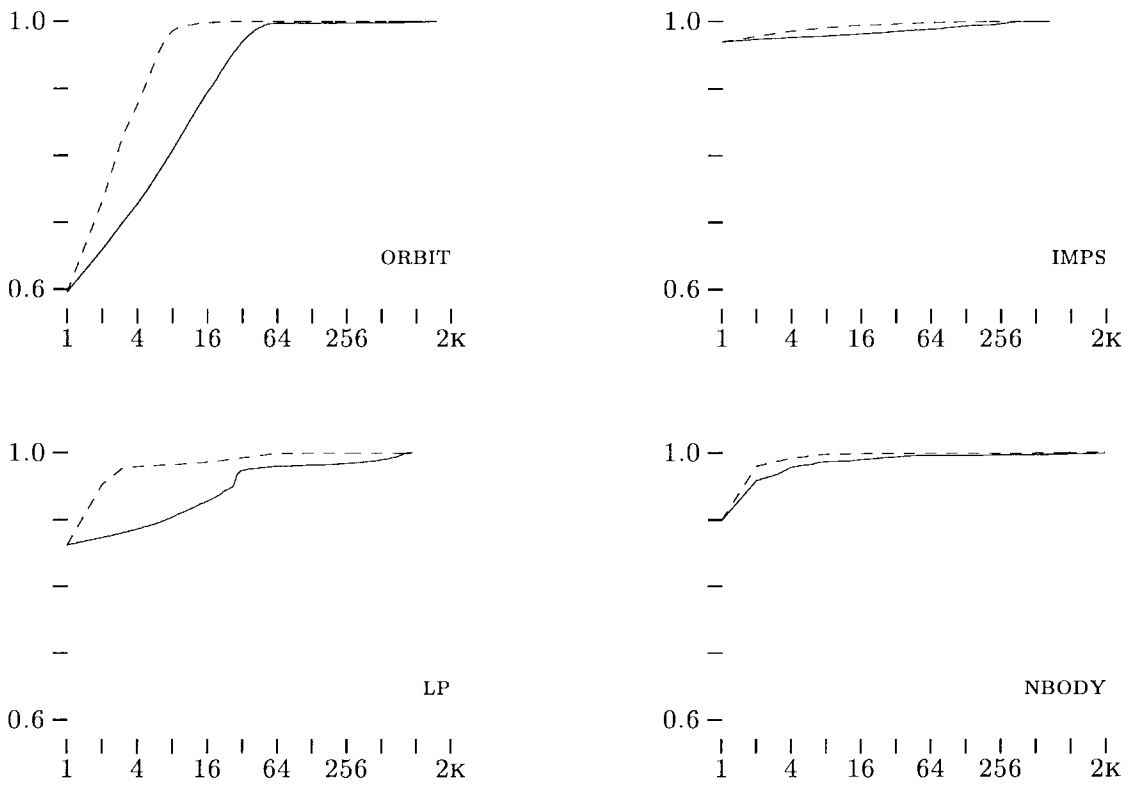


The solid lifetime curve is similar to those in the previous graph, but block lifetimes are here measured in 64KB-cache allocation cycles instead of data references. A point on the dashed active-cycle curve shows, in its y value, the fraction of dynamic blocks whose 64KB-cache active-cycle counts are no greater than its x value.* Every dynamic block is active in at least one cycle, namely its initial allocation cycle, so the two curves start at the same point.

The early, steep rise in the active-cycle curve, as compared to the lifetime curve, implies that even though there are a substantial number of blocks with lifetimes that span many allocation cycles, nearly all of these blocks are only active in a few cycles. For example, about 95% of all dynamic blocks have lifetimes of no more than 80 cycles, but the same fraction is active in no more than 4 cycles.

*Unlike the lifetime distributions in the previous graph, the bin size in these distributions is 1.

For the other test programs, the difference between the two distributions is smaller:



The differences between the curves for these programs are smaller because the programs have fewer long-lived dynamic blocks to begin with. In each case, however, nearly all multi-cycle blocks are only active in a few allocation cycles.

To summarize the behavior of dynamic memory blocks: The cyclic sweep of the allocation pointer distributes dynamic blocks both spatially, throughout the cache, and temporally, within each cache block. Most, and sometimes nearly all, dynamic blocks are one-cycle blocks and therefore cannot interfere with each other; if no other blocks interfere with them, then they will be allocated, live, and die entirely in the cache. The remaining multi-cycle blocks are dispersed in the cache by the allocation pointer, and so are unlikely to collide with each other. Nearly all multi-cycle blocks are only active in a few allocation cycles, which limits the extent to which they can interfere with other dynamic blocks or with blocks in other areas. Finally, the interference created by dynamic blocks is further limited by the fact that most dynamic blocks, regardless of lifetime, are referenced just a few times.

This section has analyzed the behavior of the memory area containing most of the blocks in each program; the next three sections consider the memory areas that account for most references.

reference counts meet or exceed this threshold are to the left of the cross in each curve above.

The high vertical positions of the busy-block crosses in all of the curves show that busy blocks account for many, and sometimes nearly all, program references. In the test programs there are between 59 and 155 busy blocks; thus busy blocks are but a small fraction of all active memory blocks:

	ORBIT	IMPS	LP	NBODY	GAMBIT
Active blocks	1,554,207	678,048	964,333	1,998,866	1,757,276
Busy blocks	119	110	155	59	104
Static	82	56	17	43	32
Loaded	1	20	48	12	18
Stack	36	34	90	4	54

There are no busy dynamic blocks in these programs, but there is no reason why other programs could not have such blocks. ORBIT has just one busy loaded block, but then ORBIT has few loaded blocks anyway.

3.6. Static and loaded blocks

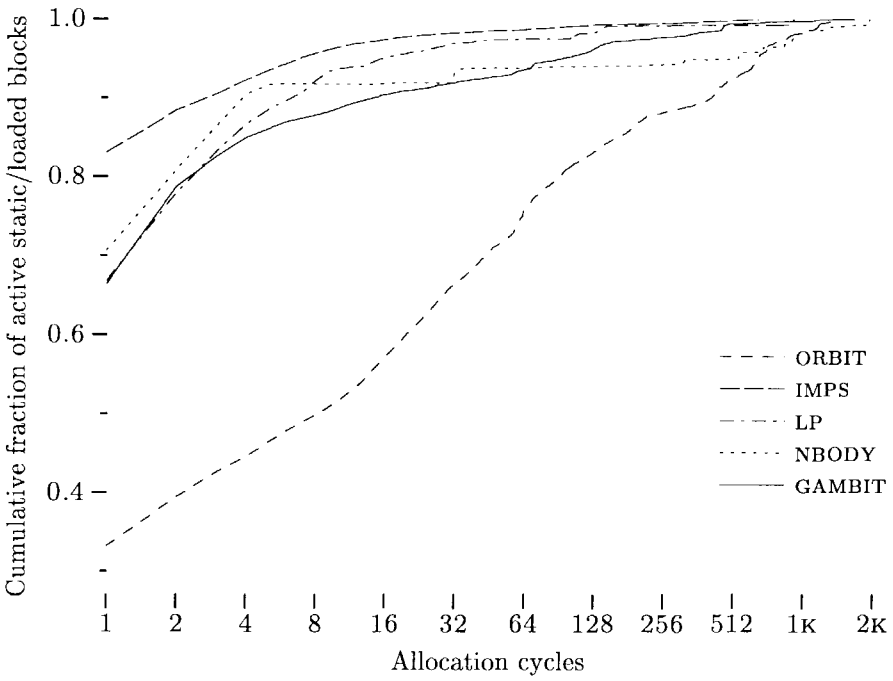
The static and loaded-data areas contain similar sorts of objects with similar behaviors, so they will be analyzed together. Static/loaded blocks are arranged within their respective areas in an essentially random fashion, so they are uniformly distributed throughout the cache. Thus one static/loaded block is about as likely as any other to collide with some other block.

A few static/loaded blocks are busy blocks. Most busy blocks probably contain closures for frequently-called procedures, but a few are artifacts of the T system itself. In all of the test programs, the few busiest static/loaded blocks contain a 49-word vector internal to the T runtime system. This vector accounts for an average of 6.7% of all program references. It contains the allocation pointer and the *limit pointer*, which points to the word immediately following the dynamic area.* In every allocation action, the allocation and limit pointers are read and the allocation pointer is written. The runtime-system vector also contains pointers to frequently-called internal routines; among these are routines for object allocation and for uncommon types of procedure calls. The pointers to the procedure-call support routines are sometimes more frequently referenced than the allocation and limit pointers; this is the case for ORBIT, IMPS, and GAMBIT. Compared to the

*When the garbage collector is enabled, a collection is triggered whenever the requested allocation would cause the allocation pointer to reach or exceed the limit pointer.

allocation and limit pointers and the internal-routine pointers, the other words in the runtime-system vector are seldom referenced.

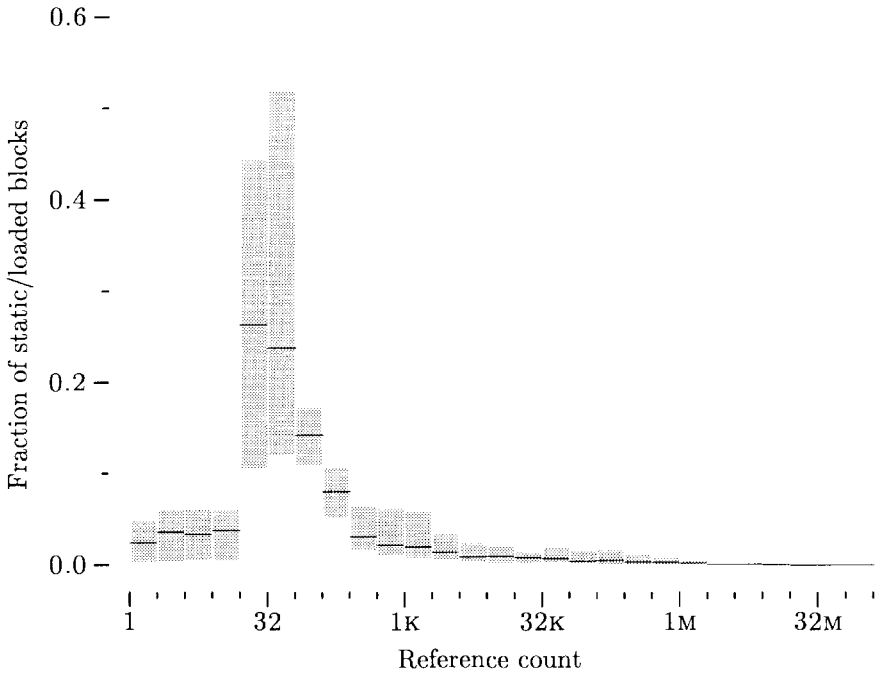
Nearly all static/loaded blocks are not busy blocks, and their behaviors are similar to those of multi-cycle dynamic blocks. Most static/loaded blocks are only active in a few 64KB-cache allocation cycles, as is shown by the cumulative frequency distributions of their active-cycle counts:



This graph is much like the separate graphs of active-cycle curves for multi-cycle dynamic blocks presented earlier (p. 32), except that lifetime curves are not shown and the curves for all programs are presented in a single graph.

Compared to the active-cycle curves for multi-cycle dynamic blocks, one curve for the static/loaded blocks starts at a lower point and does not grow as quickly. Thus, in that program, ORBIT, static/loaded blocks tend to be active in a few more cycles than multi-cycle dynamic blocks. Even in ORBIT, however, the activity of most such blocks is confined to a small part of the total program running time. Therefore most static/loaded blocks cannot be a significant source of interference in any of the test programs.

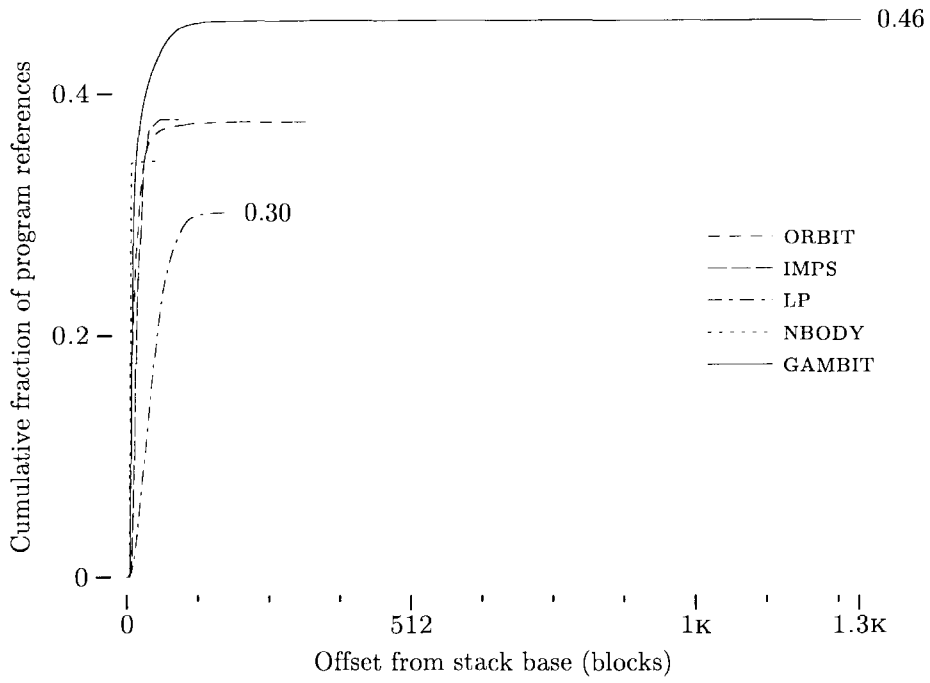
Most static/loaded blocks are not referenced many times, which also limits the amount of interference they can create:



Like multi-cycle dynamic blocks, most static/loaded blocks are referenced fewer than 64 times. In contrast, however, a significant number of blocks have reference counts greater than 1K. Because there are so few blocks in the higher sample bins, the solid average line is not visibly raised above zero.

3.7. Stack blocks

In all of the test programs, the stack exhibits high temporal and spatial locality. Nearly all stack references are concentrated in a small contiguous group of extremely busy blocks:



This graph shows, for each program, the cumulative distribution of stack references over stack block addresses. The x axis is calibrated in 64-byte blocks from the base of the stack, which is placed at the origin. A point on a curve shows, in its y value, the fraction of all program references that are to the stack block at its x value or to any stack block logically below that block. The x value of the endpoint of each curve is the maximum stack size, while the y value of the endpoint is the fraction of all references that are stack references.

The sharp upward jump at the start of each curve indicates the presence of a few extremely busy blocks at the lower end of the stack; the nearly horizontal final segments imply comparatively little activity in higher blocks in the stack. Thus the call stack is only rarely deeper than about one hundred blocks.

In most of the test programs, the stack fits entirely within the cache. As can be seen in the graph, the stack in one program, GAMBIT, reaches 1.3K blocks, while the others remain well under 1K. Thus stack blocks interfere with each other only in GAMBIT, and then only in the two smallest caches being considered, namely 32 and 64KB; in no case do busy stack blocks interfere with each other.

3.8. From behavior to performance

Having described various properties of the test programs' memory behaviors in terms of each memory area, a connection can now be made from these properties to measured cache performance. In order to do this, consider how the activity that takes place in a single cache block determines that block's independent, *local* performance.

A cache block will see references to one new dynamic memory block at the start of each allocation cycle; this block is likely to be a short-lived, one-cycle block. A cache block may also see references to a few multi-cycle dynamic blocks, a few non-busy static/loaded blocks, and perhaps a non-busy stack block; these blocks are likely to be active in only a small number of allocation cycles. All of these memory blocks, short-lived or otherwise, are non-busy, and so will be referenced relatively few times.

A cache block might also see references to one or more busy blocks. The number of busy blocks that map to a cache block places it, roughly, at either end of a range of local performance:

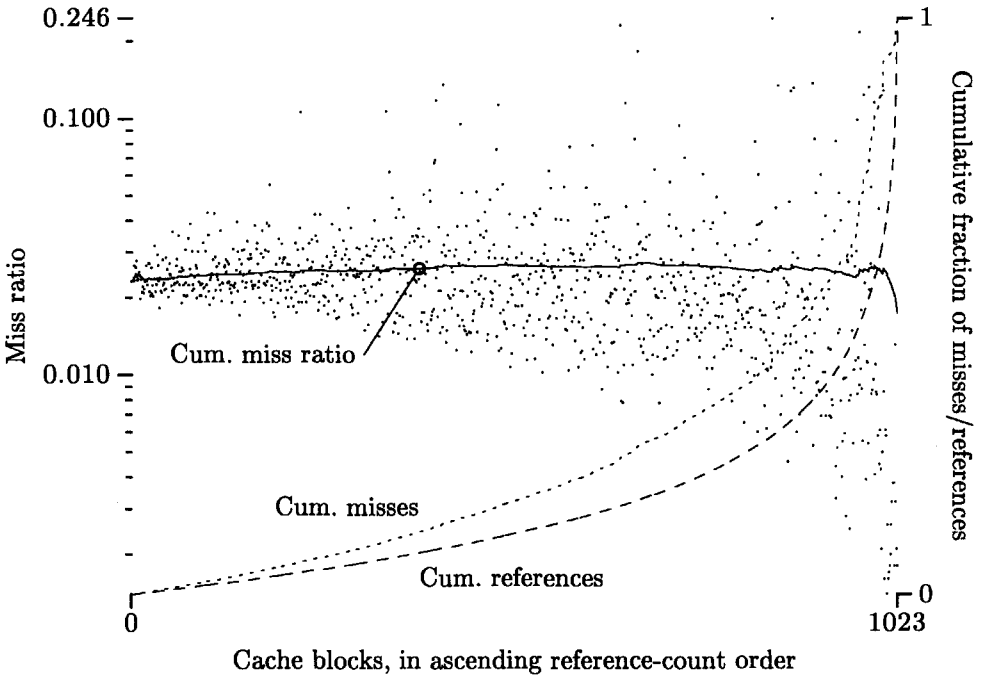
> Worst case: Two or more busy blocks map to the cache block. Because busy blocks are so frequently referenced, they may thrash, making the cache block's local performance quite bad. Thrashing memory blocks are visible as horizontal stripes in cache miss plots (p. 26).
>
> Best case: Exactly one busy block maps to the cache block. Every reference to another memory block might entail two misses, namely one to reference the other block and another to restore the busy block. But the sheer number of references to the busy block will generate enough hits to far outweigh these misses, so the cache block's local performance will be very good.

There are few busy blocks relative to the number of cache blocks, even in a small cache, so they are unlikely to collide and thrash. Moreover, it is often the case that many busy blocks are in the stack area, where they do not collide. Therefore the best case is expected to be more common than the worst case.

Most cache blocks will not have any busy blocks mapped to them. Each such less-referenced cache block will see references only to non-busy blocks, which are not likely to create much interference. Its local performance should therefore fall between the best and worst cases above. The performance of a less-referenced cache block may approach that of the worst case, but it cannot be better than the best case. Less-referenced cache blocks have too few references, and therefore too few hits, to compete with those that contain exactly one busy block.

The effect of a cache block's local performance on the overall *global* performance of the cache depends upon the total number of references that it sees. Thus most cache blocks, accounting for relatively few references, will have a small effect on the cache's global performance; the worst- and best-case cache blocks will play a much more significant role. The positive effect of the best cases should more than outweigh the negative effect of the worst cases.

The local performance of cache blocks, the relationship between local and global performance, and the balancing of worst- and best-case cache blocks is illustrated in the following graph of cache activity in ORBIT:



In this graph, the 1024 cache blocks of a 64KB, 64B-block cache are arranged on the x axis in ascending reference-count order; the least-referenced cache block is on the left, while the most-referenced cache block is on the right.

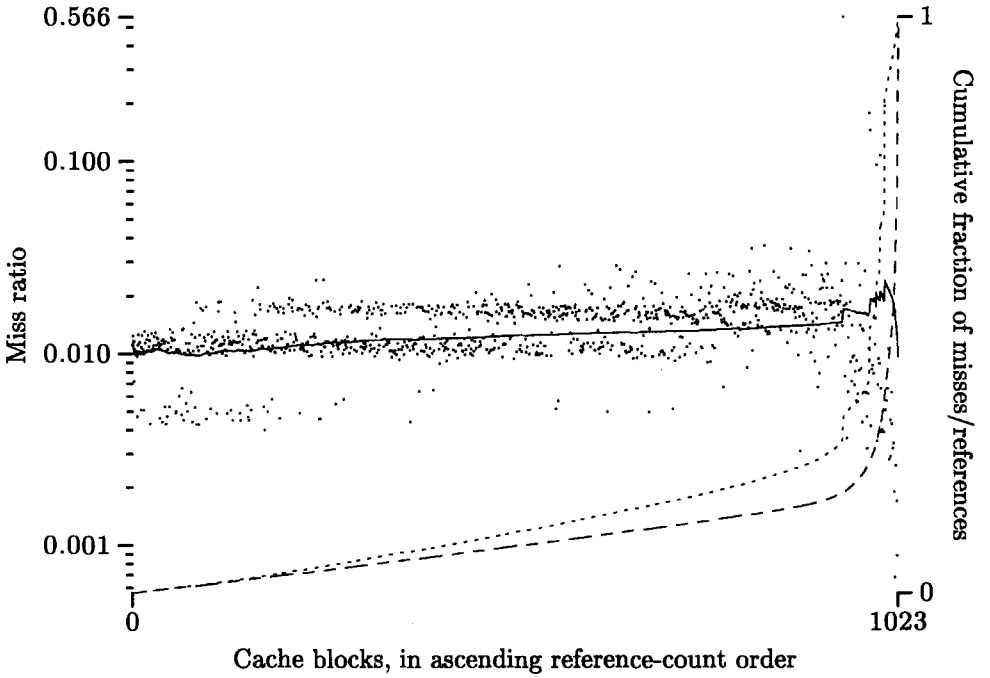
The dotted and dashed curves, associated with the right-hand scale, are the cumulative distributions among cache blocks of misses and references, respectively. As in Chapter 2, only restoration misses are shown; allocation misses are ignored. A point on the dotted curve indicates, in its y value, the fraction of all misses that occur in the x^{th} least-referenced cache block or to cache blocks referenced no more times than that block; similarly, the dashed curve accumulates cache-block reference counts. The reference and miss curves grow quickly only toward the right-hand side of the graph; thus, unsurprisingly, most misses occur in the most-referenced cache blocks.

The dots are associated with the left-hand logarithmic miss-ratio scale. There is one dot for each cache block; its height records the local miss ratio of that block. Dots in the upper quarter of the graph have bad local performance, while those in the lower quarter have good local performance. Some of the less-referenced cache blocks perform badly, but the local miss ratios of most of these blocks fall into the central half of the graph. The hundred or so most-referenced cache blocks have local miss ratios ranging from very bad, corresponding to the worst case, to very good, corresponding to the best case.

Finally, the solid cumulative miss-ratio curve shows the significance of each cache block's local performance to the global performance of the cache. A point on this curve indicates, in its y value, what the miss ratio of the cache would be if only cache blocks at and prior to its x value were being considered. The change in the y value at some point, relative to that of the preceding point, reflects the effect of the local performance of the cache block at that point upon the cache's global performance. The height of the endpoint of the curve is the global miss ratio of the cache.

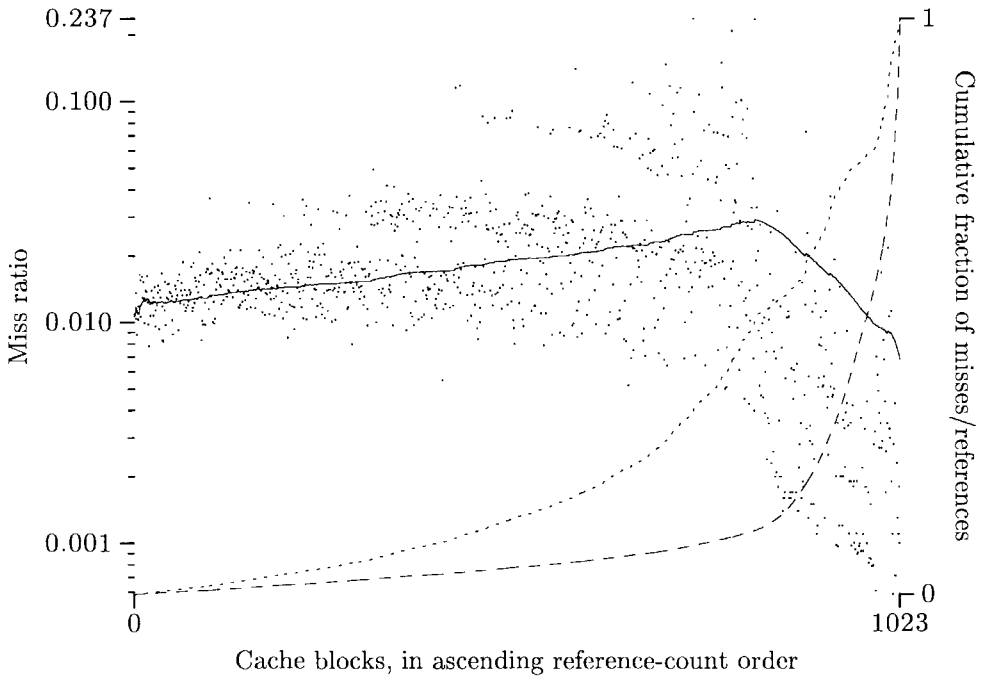
Because most cache blocks do not account for many references, the cumulative miss ratio does not change significantly until it reaches the more-referenced cache blocks. At that point, however, it becomes more volatile, with worst- and best-case cache blocks pulling it up and down, respectively. The best-case cache blocks prevail in the end, pulling the cumulative miss ratio down and more than making up for the worst cases. Because of the logarithmic miss-ratio scale, the final drop in the curve may appear small, but in fact it falls from 0.027 to 0.017, a factor of about 1.6.

A similar pattern usually holds, with variations, for the other test programs. For example, NBODY exhibits a more pronounced concentration of references and misses in the last few cache blocks, but again the best cases prevail:

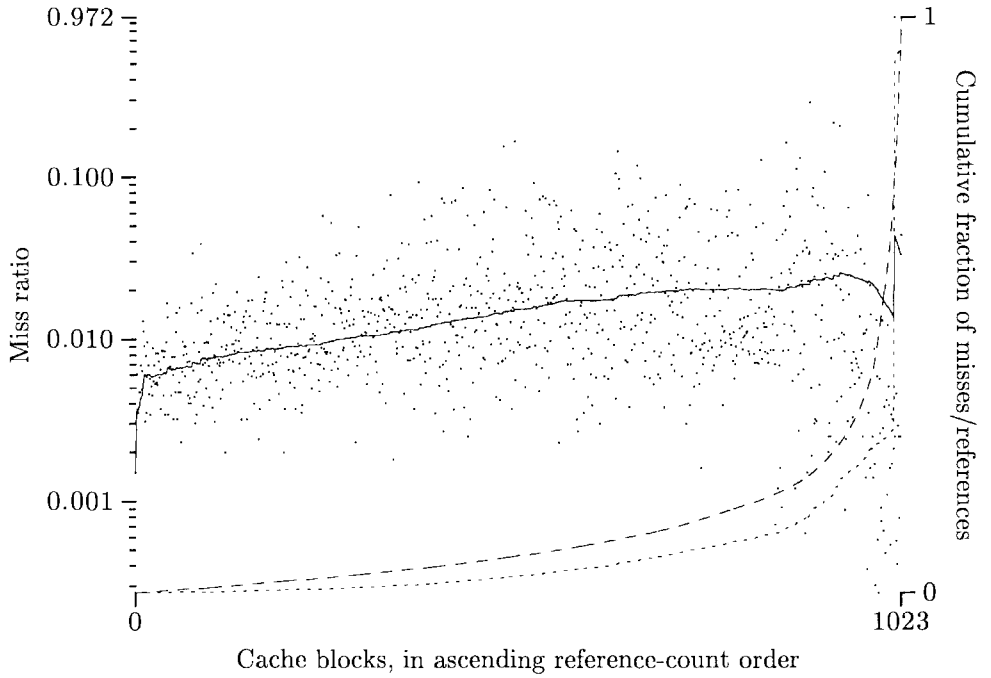


One cache block performs quite badly in NBODY, with a local miss ratio of 0.566, but its effect on the global miss ratio, while visible, is small because it does not see a large number of references.

In some programs there are no worst-case cache blocks, so the best-case blocks just improve the cache's global performance over that of the less-referenced cache blocks. This occurs in LP:

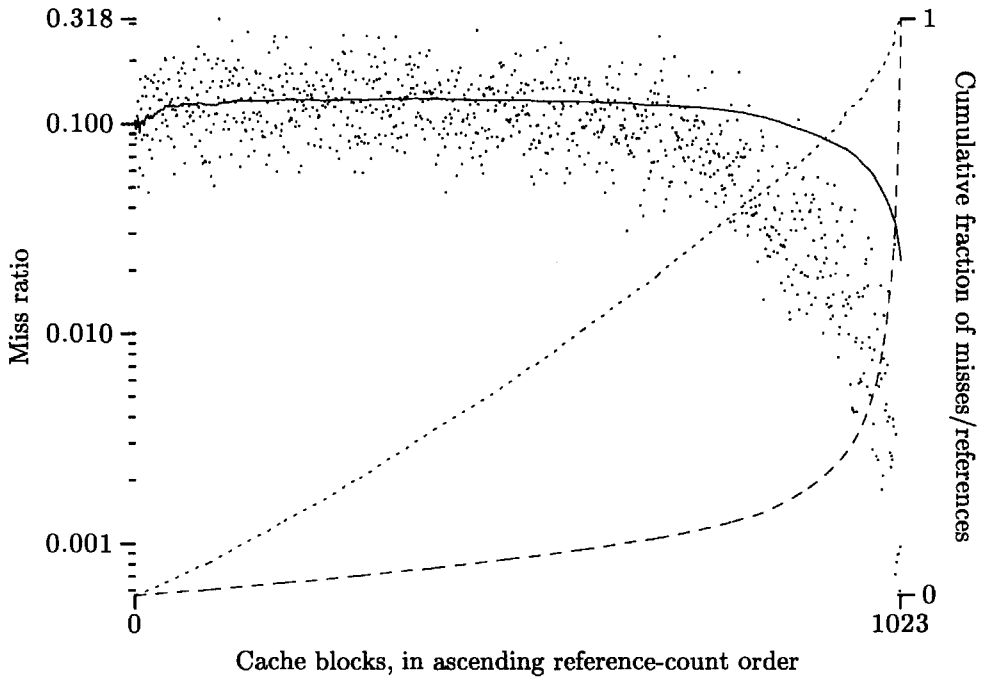


An exception to the pattern occurs in IMPS:



The initial jump in the cumulative miss-ratio curve for IMPS shows that the few least-referenced blocks have very good local performance, but these blocks account for so few references that they have only a minuscule effect on the global miss ratio. More importantly, the large jump near the end of the cumulative miss-ratio curve is caused by a single cache block with a truly terrible local miss ratio of 0.972. This cache block sees references to a busy stack block and a busy loaded block in almost perfect alternation, resulting in vigorous thrashing. The global miss ratio could be improved dramatically by eliminating this thrashing; techniques for doing so will be presented in Chapter 4.

A more notable exception to the pattern occurs in GAMBIT:



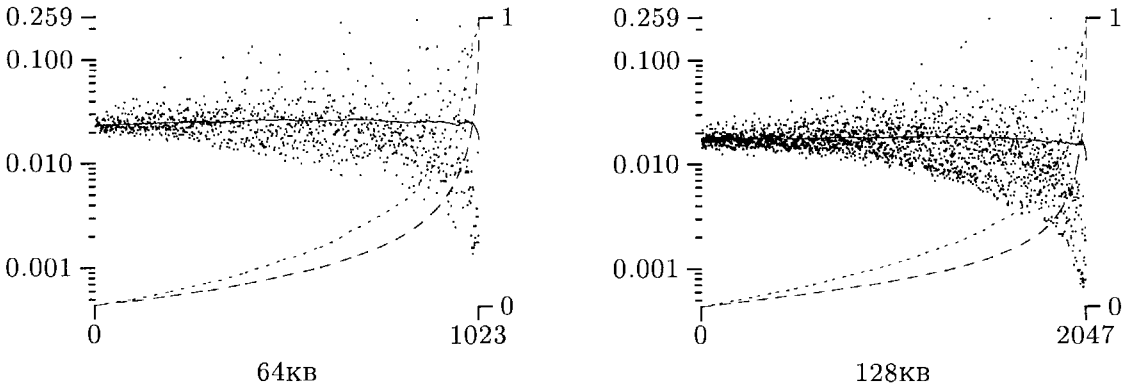
Like LP, there are no worst-case cache blocks in GAMBIT. Most of the less-referenced blocks, however, perform badly, with typical local miss ratios roughly an order of magnitude higher than those seen in the other programs. As a consequence, the dotted cumulative-miss curve for GAMBIT is nearly diagonal, indicating that misses are spread throughout the cache rather than being concentrated in the most-referenced cache blocks. In the end, though, the best-case cache blocks again pull the global miss ratio down to a more satisfactory level.

The local miss-ratio data for GAMBIT show that less-referenced cache blocks do not always have good local performance. In GAMBIT, this may be due to the fact that the program has many long-lived dynamic blocks. The dynamic-block lifetime curve for GAMBIT (the solid curve in the graph on p. 30), contains a significant jump between 16M and 32M references. If these long-lived blocks are referenced around the same time, perhaps in a linear fashion as GAMBIT produces object code in its final assembly phase, they could cause many cache misses.

The preceding graphs show the activity of the test programs in small, 64KB caches. As was seen in Chapter 2, the cache performance of the test programs improves dramatically as the cache size increases. With more cache blocks, busy memory blocks are less likely to collide; thus the worst case becomes less common and the best case becomes more common. Also, allocation cycles double (approximately) in length each time the cache size doubles; thus even more dynamic

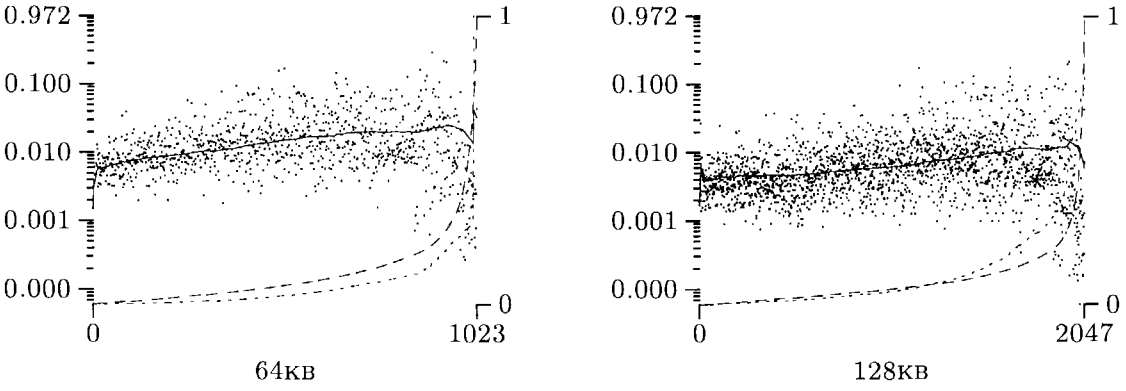
blocks will tend to be one-cycle blocks, improving the local performance of the less-referenced cache blocks.

These trends can be seen by comparing the cache-activity graphs for ORBIT running in 64KB and 128KB caches:



In the larger cache, more of the most-referenced cache blocks have good local performance. The performance of the less-referenced cache blocks is also improved, as they are more tightly clustered about the cumulative miss-ratio curve, which is lower than that of the smaller cache.

IMPS, which thrashes in a 64KB cache, exhibits a dramatic improvement in a 128KB cache:



Not only has thrashing been eliminated, but, as with ORBIT, both the less- and most-referenced cache blocks exhibit improved local performance.

The analysis has only considered 64-byte blocks. Most objects in Scheme programs are smaller than even 16-byte blocks. If smaller blocks were used, the objects responsible for causing blocks to be long-lived or busy would affect a smaller fraction of all memory blocks. Thus, performing the behavioral measurements at a smaller block size should reveal stronger extremes in various properties. For

4. Applications and extensions of the analysis

Thus far, only five Scheme test programs have been studied. Chapter 2 established that they have good cache performance without any garbage collection at all, and that a simple compacting collector performs well in most cases. Chapter 3 established that the programs' good cache performance follows from natural properties of their memory behavior.

This chapter applies the results of Chapter 3 to devise ways in which the performance of the test programs might be improved; it then goes beyond the test programs to generalize the analysis, and therefore the prior results, to other programs and other programming languages.

4.1. Improving program performance

As mentioned earlier, a simple way to improve a program's performance is to improve its cache performance by using a larger cache. The measurements and analysis of Chapter 3 suggest three other possibilities: Relocate objects to reduce interference, use the memory hierarchy more carefully, and decrease the allocation rate. None of these methods impose significant runtime costs.

Relocate objects to reduce interference. The cache-activity graph for IMPS (p. 46) and, to a lesser degree, that for NBODY (p. 44), revealed that busy-block thrashing can be a significant source of misses in small, 64KB caches. The performance of both programs could be improved by eliminating this thrashing; in terms of the cache-activity graphs, all large upward jumps in the cumulative miss ratio curves would be eliminated.

That thrashing occurs in two of the five test programs suggests that it might occur in many other programs, at least in smaller caches. While thrashing is less likely in larger caches, it is still possible. The remainder of this subsection sketches a method for reducing interference by eliminating thrashing.

The fundamental idea is to first identify all busy objects, by gathering profile information on object reference counts, and then permanently relocate these objects to new addresses so that they do not collide in the cache. While conceptually straightforward, realizing this goal requires careful engineering.

In the Scheme system used in this investigation, namely the MIPS implementation of the T system, when and how an object is relocated depends upon the type of memory in which it resides. Objects in the linked area must be relocated when the system load image is constructed. Dynamic and loaded objects must be relocated at runtime, as they are created. The stack can only be relocated once, when the system starts.

These differences suggest that the target machine's cache be divided into three regions, one for each of these groups. Arrange for the lower portion of the stack area to map into the stack's cache region; since the stack is usually small, most stack references should fall within this region. Allocate two reserved memory areas, one for linked objects and one for dynamic and loaded objects, such that each area maps exactly into its corresponding cache region. Eliminating thrashing is then a simple matter of relocating busy objects into these reserved areas in a linear fashion. There are usually few busy objects, so even in small caches it should be possible to choose sufficiently large regions. Regions chosen for a small cache will work in all larger caches, so this optimization need not be performed for every expected cache size.

It seems likely that some linked objects in the Scheme library and runtime system will be busy in most programs; thus it may suffice to relocate them just once, based upon profile data gathered from a large set of programs. For a specific program, dynamic and loaded objects can be relocated by means of a special allocation primitive that allocates objects in the appropriate reserved memory area. Calls to the ordinary allocator must be replaced by calls to this special allocator wherever busy objects are created.

This method can be taken further to not only eliminate thrashing, but to reduce the remaining interference caused by busy objects. To do so requires that the profile data also include information about the times at which each object is referenced. Then busy objects can be relocated so that those that are referenced together tend to be placed in the same memory block. As this enhancement requires significantly more profile data, it may only be worth doing for linked data objects.

The idea of using profile information to statically relocate data so as to improve program performance in a memory hierarchy was first explored by Hatfield and Gerald in the context of virtual-memory systems [26]; Stamos studied similar methods for relocating objects in Smalltalk [67]. McFarling, Pettis and Hansen, and Samples have described profile-based methods for statically improving instruction-cache performance [44, 54, 63], and at least one contemporary commercial compiler system provides a tool for that purpose [47].

Use the memory hierarchy more carefully. A second way to improve program performance is to seek ways in which the language implementation—*i.e.*, the compiler and runtime system—might make more efficient use of the memory hierarchy. The T system provides a good example of an opportunity for this method.

As mentioned in §3.6, a few words in an internal runtime-system vector account for 6.7% of all references, on average, in the test programs. The ORBIT compiler uses only 27 of the 32 registers in the MIPS architecture; three of the

remaining five are reserved for the operating system and the C library, leaving two registers free. It would be straightforward to place two of the busy runtime-system words in these registers. For example, assigning the allocation pointer and the limit pointer to registers would speed up every allocation operation by at least 3 instructions, since it would no longer be necessary to load both pointers and update the allocation pointer. The improvement would be even more significant if eliminating these memory references were also to reduce the amount of interference created by references to the runtime-system vector.

Counter-intuitively, this improvement could actually increase both miss ratios and cache overheads. Miss ratios are likely to increase because references to the allocation and limit pointers probably hit in the cache most of the time; removing these references will decrease both the total number of misses and, to a greater degree, the number of references. Cache overheads may increase because, while a program will have fewer misses, it will also execute fewer instructions. Nonetheless, overall performance, as measured by total program running time, should improve.

Decrease the allocation rate. The analysis of memory behavior showed that having a large number of one-cycle dynamic blocks is important for good cache performance. Absent other interference, one-cycle blocks will be allocated, live, and die entirely in the cache. The number of one-cycle blocks can be increased by increasing the lengths of allocation cycles, which can be done by decreasing the allocation rate. While the allocation rate could be decreased by using a more imperative programming style, a better approach is to use a compiler that employs static-analysis methods to enable optimizations that decrease the allocation rate.

One way that a compiler can decrease the allocation rate is to convert heap allocations into stack allocations. This is already done to some extent by the T system's compiler, ORBIT, which analyzes procedure closures to determine when they can be allocated on the stack instead of in the heap [39]. Good Lisp compilers use stack allocation for floating-point numbers [11, 68]; a more general analysis can be used to stack-allocate other small objects [13, 51].

Stack-allocation optimizations must be used with care: The number of cache misses incurred while referencing objects in the stack must not be larger than the number of misses avoided by increasing allocation-cycle lengths. There is no way for a compiler to guarantee this property, since that requires knowing too much about the runtime behavior of the program being compiled. In practice, however, judiciously used stack-allocation optimizations usually yield a stack that tends to be resident in the cache. In fact, the cache misses avoided by referencing cache-resident stack-allocated objects that would otherwise be in the heap may be more significant than the misses avoided by increasing allocation-cycle lengths.

For example, the stack optimizations performed by ORBIT work well in a cache. It was seen in §3.7 that the stack has very high spatial and temporal reference locality in ORBIT-compiled programs, since most stack activity occurs in a few busy stack blocks. As long as these busy blocks do not thrash, they will tend to reside in the cache. If these blocks were not busy, however, and yet were referenced with moderate frequency over long periods of time, then they would not tend to be cache-resident; the cyclic sweep of the allocation pointer would flush them once per allocation cycle. Therefore stack-allocation optimizations should only be used when there is reasonable certainty that the stack will retain its high reference locality.

The ability of carefully used stack-allocation optimizations to improve cache performance by converting heap allocations into stack allocations leads to:

Conjecture 1. Stack allocation can be faster than garbage collection.

Appel has argued for the reverse proposition, namely that garbage collection can be faster than stack allocation [3]. Appel observes that the overhead of a copying garbage collector depends not upon the amount of garbage it reclaims, but upon the amount of live data it must copy during each collection. Since this latter quantity is approximately constant in many programs, collection overhead can be made arbitrarily small by increasing the amount of physical memory available for the heap, which will decrease the collection frequency. The overhead of managing a stack, in contrast, is fixed. Stack frames are usually not large, and there is no way to avoid the instructions that push and pop each frame. Therefore garbage collection can be made cheaper than stack allocation by using more physical memory.

Appel's argument assumes that the cost of accessing memory is constant, but this is not true in memory systems with caches. If cache misses have a significant cost, the instructions required to manage a stack that tends to reside in the cache may be cheap compared to the misses incurred while referencing objects that reside in the heap.

A case in point is Appel's implementation of Standard ML [5]. Any implementation of ML is likely to have high allocation rates, if only because ML encourages a highly-functional programming style [52]. In Appel's implementation, allocation rates are made higher still by his decision to allocate procedure-call frames in the heap rather than on a stack. This strategy has a number of advantages. Heap-allocating call frames simplifies the compiler, which need not analyze closures to determine when they can be stack-allocated. Using heap-allocated call frames also admits a particularly simple implementation of the call-with-current-continuation primitive; with a more conventional call stack, intricate support from the runtime

system is required for this operation to be efficient [15, 29]. Despite these advantages, Conjecture 1 suggests that the cache performance of Appel's ML system might be improved considerably by modifying the compiler to allocate call frames on a stack.

Aside from stack-allocation optimizations, a second way to decrease the allocation rate is to eliminate some heap allocations altogether. This goal has been investigated in the context of pure functional languages [24, 73], and could perhaps be adapted to languages such as Scheme and ML, which tend to be used in a functional style.

This section has shown three ways in which the performance of the five Scheme test programs might be improved; the results of the next section will make these methods applicable to a wider range of programs.

4.2. Generalizing the analysis

In Chapter 3, the memory behaviors of the test programs were analyzed to explain why their cache performance is so good. In this section, the analysis will be generalized to other programs and to other languages.

According to the analysis of Chapter 3, a program will have good cache performance if it satisfies the following three properties:

(1) The program has few busy memory blocks;

(2) The program has many short-lived dynamic blocks that are referenced just a few times;

(3) Nearly all other blocks are only active in a few allocation cycles, and are also referenced just a few times.

None of these properties are specific to the test programs, therefore they should hold for other Scheme programs written in a similar, mostly-functional style. Moreover, none of these properties are specific to Scheme, hence:

> **Conjecture 2.** Properties (1)–(3) above hold for programs written in a mostly-functional style in garbage-collected languages other than Scheme.

Reasoning about other programming styles leads to:

> **Conjecture 3.** Properties (1)–(3) above hold across a range of programming styles in garbage-collected languages.

This conjecture is based upon the intuitive observation that, across programming styles, allocation rates vary inversely with object lifetimes.

Recall that the number of one-cycle dynamic blocks depends upon the cache size, the allocation rate, and the lifetimes of dynamic blocks; the lifetimes of dynamic blocks, in turn, depend upon the lifetimes of the objects in those blocks. In a programming language that encourages a more functional style than Scheme, *e.g.*, ML, the allocation rate is higher, but object lifetimes are shorter. Therefore ML programs may also have a large number of one-cycle dynamic blocks.* There is no reason to believe that ML programs will have substantially more busy blocks, and the highly-functional style suggests that there will be even fewer multi-cycle non-busy blocks. Thus it is plausible that properties (1)–(3) will hold for ML programs.

Toward the other end of the functional–imperative spectrum, in a programming language that encourages a more imperative style than Scheme, *e.g.*, CLU, object lifetimes are longer, but allocation rates are lower. Therefore CLU programs may also have a large number of one-cycle dynamic blocks. In this case, however, it is less clear that properties (1) and (3) will hold. The more imperative style suggests that CLU programs may have more multi-cycle, non-busy blocks that are active in many cycles. Thus the sharp distinction between busy and non-busy blocks that was observed in Chapter 3 may not be seen in CLU programs. Nonetheless, properties (1)–(3) may hold in languages that encourage a slightly more imperative style than Scheme.

4.3. Programming styles and cache performance

These considerations of programming style lead to a final conjecture:

Conjecture 4. Allocation can be preferable to mutation.

That is, on machines where cache performance can significantly impact program performance, the performance of programs written in a mostly-functional style in a linearly-allocating, garbage-collected language may be superior to that of programs written in an imperative style in a language without garbage collection.

The intuitive argument for this conjecture is as follows. Allocation activity is like a wave that continually sweeps through the cache; the allocation pointer defines the crest of the wave. A program written in a mostly-functional style rides the allocation wave, just as a surfer rides an ocean wave. The program loads data from old dynamic blocks in front of the wave's crest; there is a good chance that these blocks are still in the cache, since the vast majority of dynamic blocks

*It may be necessary to stack-allocate procedure-call frames in order to achieve this.

are one-cycle blocks and there is little interference from most other blocks. The program then computes on this data and stores the result, usually in new dynamic blocks just behind the crest; it is highly likely that these blocks are still in the cache, since they were just allocated.

The work involved in copying data from old dynamic blocks to new ones, together with the accompanying cost of eventually running a garbage collector, may seem wasteful. An imperative style, however, will have other costs. In place of garbage-collection costs will be the costs of allocation and deallocation operations, which could be significant in terms of both instructions executed and cache misses incurred.

More importantly, a program written in an imperative style will not benefit from the naturally good cache performance that is implied by properties (1)–(3) above. In an imperative program, whether two data objects interfere in the cache, or even thrash, is usually a matter of chance. There are static-analysis methods for improving the cache performance of specific types of imperative programs; *e.g.*, an optimization called *blocking* can improve the cache performance of matrix computations [25, 41]. It seems unlikely, however, that methods will be found for improving the cache performance of a wide class of imperative programs, especially a class that includes programs that make use of many small and short-lived data objects.

Proponents of garbage-collected programming languages have long argued their case from standpoints of correctness and programmer productivity. It is plausible that such languages may also have a significant performance advantage on machines where cache performance is an important part of program performance.

4.4. Conclusion

This chapter has applied the measurements and analysis of Chapter 3 to show three ways in which the cache performance of the test programs could be improved. These methods involve relocating objects to reduce interference, making more careful use of the memory hierarchy, and decreasing the allocation rate; none of these optimizations impose significant runtime costs.

This chapter has also extended the analysis of Chapter 3, arguing that it should apply to other Scheme programs, to programs written in different languages but in a similar style, to programs written in languages that encourage an even more functional style, such as ML, and perhaps even to languages that encourage a somewhat more imperative style, such as CLU. There are, therefore, good reasons to believe that the results of Chapter 2, as well as the performance-improvement methods discussed in this chapter, are widely applicable.

5. Summary and conclusion

Inexorable trends in computer technology are making cache performance an increasingly important part of program performance. Prior work on the cache performance of garbage-collected languages has either assumed or argued that programs written in these languages will have poor cache performance if little or no garbage collection is done. This dissertation has argued to the contrary: Many such programs are naturally well-suited to the direct-mapped caches typically found in high-performance computer systems. This conclusion is supported by measurements of the cache performance of five nontrivial Scheme programs, by a qualitative analysis of how the programs' memory behaviors determine their cache performance, and by considerations of how programming style determines memory behavior.

The control experiment, reported in Chapter 2, revealed that the programs have excellent cache performance without any garbage collection at all. On two hypothetical processors, one slow and one fast, each coupled with caches of typical sizes and a realistic high-performance memory system, the programs spend less than five percent of their total running time, on average, waiting for cache misses to be serviced. With so little room for improvement, seeking better cache performance hardly seems necessary; human effort might be better spent improving other performance aspects of the hardware, the language system, or even the programs themselves. Moreover, no method for improving cache performance that imposes significant runtime costs of its own could possibly be effective. Aggressive garbage collection is likely to be such a method.

In practice, some garbage collection must be done in order to ensure good virtual-memory performance. The second experiment described in Chapter 2 showed that when the test programs are limited to a modest amount of memory, they perform well, in most cases, with a simple, efficient, and infrequently-run compacting collector. In the remaining cases, they should perform well with a simple and infrequently-run generational compacting collector. An infrequently-run compacting collector yields good program performance because it can collect often enough to minimize virtual-memory page faults, yet rarely enough to keep garbage-collection overhead low. Moreover, by collecting infrequently and allowing objects to be allocated linearly from a large contiguous memory area, it approximates the idealized case of no collection and thereby takes advantage of the program's naturally good cache performance.

In Chapter 3, a connection was established between the memory behavior of the test programs and their measured cache performance when run without garbage collection. The mostly-functional programming style typically used in Scheme programs implies that most data objects, and therefore most memory blocks, have

very short lifetimes and are referenced only a few times. The linear sweep through memory of the allocation pointer naturally disperses these short-lived blocks in time and space so that they rarely interfere in the cache. Most blocks with longer lifetimes can create only a limited amount of interference, for they are active in only a few allocation cycles and, like short-lived blocks, are referenced only a few times. The few long-lived blocks capable of significant interference, the busy blocks, are referenced so often, and collide in the cache sufficiently rarely, that they more often improve, rather than degrade, overall cache performance. Therefore the test programs have good cache performance because they are naturally well-suited to direct-mapped caches.

Even though Chapter 2 showed that the cache performance of the test programs is not desperately in need of improvement, Chapter 4 described three methods by which it might be improved anyway. While some of these techniques are not simple to implement, all are practical and none require a sophisticated garbage collector or impose significant runtime costs.

Finally, Chapter 4 reconsidered the behavioral properties leading to good cache performance that were identified in Chapter 3. It was argued that these properties should hold for other Scheme programs and for programs written in different languages but in a similar, mostly-functional style. These properties are also likely to hold for programs written in languages that encourage a more functional style, and they may hold for programs written in languages that encourage a somewhat more imperative style.

Therefore the results of the previous chapters are likely to apply to a wide range of programs in a variety of garbage-collected languages. Such programs will have good cache performance because their memory behaviors are naturally well-suited to direct-mapped caches. The best memory-allocation strategy will be linear allocation; the best garbage-collection strategy will be one of infrequent compacting collection. Complex and costly means for improving cache performance, such as aggressive garbage collection, are likely to be neither necessary nor effective.

The remainder of this chapter reviews prior work and discusses topics for future work.

5.1. Prior work

Improving the performance of garbage-collected programming languages is a goal of long standing. In relation to memory hierarchies, this goal has motivated the design of garbage collectors that improve program performance by improving virtual-memory performance. Fenichel and Yochelson [23], elaborating upon an idea due to Minsky [46], described the first copying compacting collector designed specifi-

cally to reduce page faults. Their method was further extended by Cheney [14], who devised an elegant and significant space optimization, and by Baker [6], who invented an incremental variant of Cheney's algorithm. The next major advance was generational garbage collection, due to Lieberman and Hewitt [42], which exploits natural properties of programs to reduce collection pause times and further improve virtual-memory performance. Generational collection is now widely accepted, having been implemented in many different language systems [4, 7, 12, 18, 49, 64, 66, 69, 72].

In contrast with the work done on virtual-memory performance, only recently has serious attention begun to be paid to the cache performance of garbage-collected languages. Peng and Sohi seem to have been the first to realize that cache performance could be improved by exploiting natural properties of programs [53]. Wilson, Lam, and Moher originally suggested that a generational garbage collector might be designed specifically to improve cache performance [74, 75]; their work was taken further by Zorn [79]. Important work in this area has also been done by Koopman, Lee, and Siewiorek [36, 37]; as they are concerned with the performance of combinator-graph reduction, an implementation technique for lazy functional languages [55], their work will not be reviewed in detail here.

The remainder of this section summarizes the work of Peng and Sohi, Wilson *et al.*, and Zorn, and compares it to the investigation presented here. Some of these authors' conclusions are contrary to those of the present work, but the apparent contradictions are due either to significant differences in the classes of machines studied or to statements that are, in fact, unjustified.

The earliest published study of the data-cache performance of a garbage-collected language is that of Peng and Sohi [53], who measured eight small Lisp programs running on a simulated machine similar to the Symbolics 3600 Lisp machine [50]. As in the present work, they measured programs when run without garbage collection in order to understand intrinsic program behavior. Their measurements of dynamic memory-block lifetimes are quite similar to those shown in §3.4; they also measured inter-reference times, which were not measured here. Peng and Sohi observed the ability of a cache-block allocation instruction to improve cache performance by eliminating allocation fetches, as noted in §2.5, and measured this improvement. They also developed two further optimizations; one reduces memory traffic by eliminating write-backs of cache blocks that contain only garbage objects, while the other improves cache performance further by means of a replacement strategy that reuses blocks containing garbage objects before reusing blocks containing live objects.

Peng and Sohi's work was pioneering, but it is limited in a number of ways. Their second two optimizations achieve near-zero miss and memory-traffic ratios

for their test programs, but require both custom hardware and a collector capable of detecting garbage quickly. The latter requirement can only be met by aggressive collection or by reference-counting, which has serious drawbacks of its own [16]. Peng and Sohi did not consider the hardware and software costs of implementing such strategies. More generally, they only measured the performance of small caches, from 512B to 64KB, and they only considered references to dynamic memory, assuming that some other mechanism would cache stack references. Combined with the fact that they simulated a machine designed specifically for Lisp rather than a more conventional architecture, these limitations imply that their conclusion, namely that garbage-collected programs are likely to perform poorly with direct-mapped caches, is not widely applicable; in particular, it does not contradict the results of the present work.

Wilson, Lam, and Moher were the first to study the interactions between generational garbage collection and cache performance [74, 75]. Observing that generational collectors can improve the performance of virtual memories by cyclically reusing a modest amount of memory on a relatively small time scale, they reasoned that this idea could also be applied to improve the performance of caches. While intuitively appealing, this reasoning does not obviously hold when the fundamental differences between virtual memories and caches are considered; in particular, virtual memories typically employ a least-recently-used replacement policy, while practical caches are direct-mapped or perhaps set-associative with a small set size. Nonetheless, thus was born the notion of what has here been called 'aggressive garbage collection.'

To test this idea, Wilson *et al.* measured the miss ratios of four Scheme programs running with a generational collector in direct-mapped, two-way, and four-way set-associative caches. Their measurements show that cache performance is much better when the cache is large enough to hold most of the memory used by the collector; their data also shows that set-associative caches have better performance, in most cases, than direct-mapped caches of the same size.

Taken alone, these results are not surprising. In their conclusion, however, Wilson *et al.* state:

> If a large enough cache is available, software techniques can decrease miss ratios appreciably by keeping the youngest generation in cache, and reducing its footprint by reusing a creation region at every cycle, rather than simply alternating between two semispaces. [75, p. 41].

The experiment described in their paper does not provide evidence for this assertion. Wilson *et al.* only measured the performance of a single aggressive collector; to support the above conclusion requires comparing an aggressive collector with an infrequently-run semispace collector. Therefore the results of the present work,

including the observation that infrequent compacting collection should suffice for good cache performance, are not contradicted by the data of Wilson *et al.*

Beyond this unsupported conclusion, the work of Wilson *et al.* is limited in other ways. They ran their test programs in a byte-coded implementation of Scheme [34], rather than in a system with a true compiler. The programs themselves are relatively small, executing tens of millions of byte-code instructions and allocating several megabytes of data. Finally, Wilson *et al.* only measured miss ratios, making no attempt to account for the temporal costs of garbage collection and cache activity, and they did not consider caches with write-miss policies capable of eliminating allocation fetches.

More recently, Zorn, expanding upon results presented in his doctoral dissertation, went beyond Wilson *et al.* to compare the cache performance of two different generational garbage collectors [78, 79]. Zorn implemented both a noncompacting mark-and-sweep collector, which moves objects only when they are advanced from one generation to the next, and a more traditional copying collector. He measured the simulated data-cache performance of four large Lisp programs running with these collectors. Each program allocates between 15E6 and 82E6 bytes of data and makes tens of millions of data references, although only the first twenty million references of each program were used. Measurements were made with the collectors' youngest generations set to sizes ranging from 128KB to 2MB, so in all cases the collectors were aggressive.

Zorn's main experimental results are that larger cache sizes lead to lower miss ratios, that the mark-and-sweep collector achieved lower miss ratios than the copying collector, and that only the copying collector benefited significantly from two- and four-way set-associative caches.

None of Zorn's experimental results are inconsistent with those of the present work, but one conclusion and a related conjecture are contrary to the results presented here. Zorn states that he has shown that "even the cache locality of garbage-collected Lisp programs can be improved substantially" [79, p. 39], but his experimental data does not justify this assertion. Zorn did not perform a control experiment, running the programs without garbage collection to see whether their cache performance needed improvement in the first place, nor did he measure the performance of the programs when run with a non-aggressive collector.* From the conclusion that the cache performance of garbage-collected programs can be improved, Zorn goes on to conjecture that aggressive collection will be an effective

*Zorn claims that the cache performance of a program using a collector that is invoked after every 2MB of allocation will closely approximate the non-collection case [79, p. 19]. This claim is implausible, for it completely ignores the cost of running the collector and the potential negative effect of collection upon the program's cache state.

means for improving program performance. Since neither the conclusion nor the conjecture are justified, they do not contradict the results of the present work.

A further defect of Zorn's work is that, like Wilson *et al.*, he does not consider the temporal costs of garbage collection and cache activity; rather, he only compares the miss ratios of programs running with different collectors. Because miss ratios are only a partial characterization of program performance, this flaw undermines his comparison of collectors. It is possible for one collector to have a lower miss ratio than another but execute many more instructions, so it is essential to also measure the collectors' costs in terms of instructions executed, and to relate the miss ratios to those costs. Moreover, Zorn does not discuss the impact of allocation fetches on cache performance, and he does not consider write policies capable of eliminating allocation fetches.

In summary, then, known prior work on the cache performance of garbage-collected languages is limited in a number of ways. Among these limitations are assumptions of custom hardware, measurements of relatively small programs, and measurements of language implementations based upon interpreters rather than compilers. The more recent prior work has failed to consider cache write policies capable of eliminating allocation fetches, which are a significant fraction of all fetches in large caches. No prior work has measured program performance in a way that correctly accounts for the temporal costs of cache misses and garbage collection. Finally, and most importantly, claims made in prior work about the necessity and efficacy of aggressive garbage collection are not justified by the data presented.

5.2. Future work

The four conjectures of Chapter 4 are good starting points for further research.

The first conjecture, that stack allocation can be faster than garbage collection, can be tested by studying the performance of programs compiled by a Scheme compiler modified to heap-allocate procedure-call frames, and the performance of programs compiled by an ML compiler modified to stack-allocate call frames.

In principle, the second conjecture, that the properties of memory behavior that yield good cache performance hold for programs written in a mostly-functional style in garbage-collected languages other than Scheme, and the third conjecture, that these properties hold across a range of programming styles, are not difficult to test. In practice, however, using the tools developed in the course of this work to measure the memory behaviors of programs running in other language systems will require significant programming effort. Nonetheless, this line of investigation should yield interesting results.

The third conjecture suggests that even languages that usually require manual deallocation might benefit from linear allocation and infrequent garbage collection. Many programs in more conventional languages use the heap in a manner similar to that of Scheme programs: Most objects are small and short-lived [20, 77]. Thus a linear allocation strategy might yield superior cache performance compared to the strategies usually employed in implementations of such languages, which are typically optimized to conserve memory space and processor time [35, 38]. Linear allocation requires some sort of garbage collection; while complete collection is impossible for languages such as C, Bartlett has devised compacting collectors that require only minor program modifications [8, 9]. So an interesting experiment would be to modify a set of C programs to use Bartlett's collector and compare their performance with that of the original versions.

The fourth conjecture, that allocation can be preferable to mutation, is inherently difficult to test empirically. Such a test requires the direct comparison of different programming styles, and is thus more in the domain of software engineering than of language implementation. Perhaps the easiest way to test this conjecture is to simply wait. If the proponents of each style keep working on better compilers, then eventually one style may prove superior.

A final question is whether the cache performance of programs written in other programming styles can be explained in terms of their memory behaviors, as was done for the Scheme test programs in Chapter 3. Given a means of generating reference traces, producing cache-activity graphs similar to those shown in Chapter 3 is straightforward. Except for special classes of programs such as matrix computations, however, it is not clear that there will be a small and identifiable set of behavioral properties that suffice to explain measured cache performance.

5.3. Final exhortations

To hardware designers: A mechanism for avoiding allocation fetches, such as a write-miss policy of write-validate, is crucial to the good performance of garbage-collected languages. It is already known to be important to more traditional languages; the present work provides one more justification.

To language implementors: Non-interactive programs are likely to perform well with linear allocation and a simple, infrequently-run generational compacting collector, even when cache performance is a concern. If you must push cache performance to the limit, it is probably best to apply static methods rather than pursue complex runtime methods.

To language designers: Do not give up on garbage-collected languages that encourage a mostly-functional programming style. In the long run, as locality becomes ever more important to program performance, they may prove to have a significant performance advantage.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[2] Luigia Aiello and Gianfranco Prini. An efficient interpreter for the lambda-calculus. *Journal of Computer and System Sciences*, 23(3):383–424, December 1981.

[3] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[4] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software – Practice and Experience*, 19(2):171–183, February 1989.

[5] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[6] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[7] S. Ballard and S. Shirron. The design and implementation of VAX/Smalltalk-80. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 127–150, Addison-Wesley, 1983.

[8] Joel F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots.* Research Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, February 1988.

[9] Joel F. Bartlett. *Mostly-Copying Garbage Collection Picks Up Generations and C++.* Technical Note 12, Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, October 1989.

[10] Anita Borg, R. E. Kessler, Georgia Lazana, and David W. Wall. *Long Address Traces from RISC Machines: Generation and Analysis.* Research report 89/14, Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, September 1989.

[11] Rodney A. Brooks, Richard P. Gabriel, and Guy L. Steele Jr. An optimizing compiler for lexically scoped LISP. In *Symposium on Compiler Construction*, pages 261–275, ACM, 1982.

[12] Patrick J. Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In Norman Meyrowitz, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 119–130, ACM, September 1986.

[13] David R. Chase. Safety considerations for storage allocation optimizations. In *Conference on Programming Language Design and Implementation*, pages 1–10, ACM, June 1988.

[14] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[15] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for continuations. In *Conference on Lisp and Functional Programming*, pages 124–131, ACM, July 1988.

[16] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[17] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Conference on Lisp and Functional Programming*, pages 43–52, ACM, June 1992.

[18] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[19] Eric DeLano, Will Walker, Jeff Yetter, and Mark Forsyth. A high speed superscalar PA-RISC processor. In *IEEE Computer Society International Conference*, pages 116–121, IEEE, February 1992.

[20] John DeTreville. *Heap Usage in the Topaz Environment.* Technical Report 63, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, August 1990.

[21] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. In M. E. Stickel, editor, *Tenth International Conference on Automated Deduction*, pages 653–654, Volume 449 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 1990.

[22] Mark Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Conference on Lisp and Functional Programming*, pages 119–130, ACM, 1990.

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

[24] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, 1993. To appear.

[25] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[26] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[27] John L. Hennessey and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Palo Alto, CA, 1990.

[28] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, 24(9):18–29, September 1991.

[29] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Conference on Programming Language Design and Implementation*, pages 66–77, ACM, June 1990.

[30] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* Ph.D. thesis, Computer Science Division, University of California at Berkeley, November 1987. Available as UCB/CSD Technical Report 87/381.

[31] Norman P. Jouppi. Cache write policies and performance. In *International Symposium on Computer Architecture*, pages 191–201, IEEE, May 1993.

[32] Norman P. Jouppi, Jeremy Dion, David Boggs, and Michael J. K. Nielsen. *MultiTitan: Four Architecture Papers.* Research Report 87/8, Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, April 1988.

[33] Gerry Kane. *MIPS RISC Architecture.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[34] Richard Kelsey and Jonathan Rees. A tractable scheme implementation. Submitted for publication, September 1993.

[35] Donald E. Knuth. *Fundamental Algorithms.* Volume 1 of *The Art of Computer Programming*, Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[36] Philip J. Koopman, Jr., Peter Lee, and Daniel P. Siewiorek. Cache performance of combinator graph reduction. In *Proceedings of the International Conference on Computer Languages*, pages 39–48, IEEE, New Orleans, March 1990.

[37] Philip J. Koopman, Jr., Peter Lee, and Daniel P. Siewiorek. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, April 1992.

[38] David G. Korn and Kiem-Phong Vo. In search of a better Malloc. In *Proceedings of the Summer USENIX Conference*, pages 489–506, 1985.

[39] David A. Kranz. *Orbit: An Optimizing Compiler for Scheme.* Ph.D. thesis, Yale University, New Haven, Connecticut, February 1988.

[40] David A. Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: An optimizing compiler for Scheme. In *Symposium on Compiler Construction*, pages 219–233, ACM, June 1986.

[41] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, ACM, April 1991.

[42] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[43] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Volume 114 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1981.

[44] Scott McFarling. Program optimization for instruction caches. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, ACM, April 1989.

[45] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[46] Marvin L. Minsky. *A LISP Garbage Collector Algorithm Using Serial Secondary Storage.* Memo 58, MIT Artificial Intelligence Laboratory, Cambridge, MA, October 1963.

[47] MIPS Computer Systems, Inc. *MIPS Languages Programmer's Guide.* Sunnyvale, CA, September 1988. Order number 02-00035.

[48] MIPS Computer Systems, Inc. *MIPS R4000 Microprocessor User's Manual.* Sunnyvale, CA, 1991.

[49] David A. Moon. Garbage collection in a large Lisp system. In *Conference on Lisp and Functional Programming*, pages 235–246, ACM, 1984.

[50] David A. Moon. Architecture of the Symbolics 3600. In *International Symposium on Computer Architecture*, pages 76–83, IEEE, June 1985.

[51] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Conference on Programming Language Design and Implementation*, pages 116–127, ACM, June 1992.

[52] Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1992.

[53] Chih-Jui Peng and Gurindar S. Sohi. *Cache Memory Design Considerations to Support Languages with Dynamic Heap Allocation.* Technical Report 860, Computer Sciences Department, University of Wisconsin at Madison, July 1989.

[54] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Conference on Programming Language Design and Implementation*, pages 16–27, ACM, June 1990.

[55] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, London, 1987.

[56] Steven A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach.* Morgan Kaufmann, Palo Alto, CA, 1990.

[57] George Radin. The 801 minicomputer. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, ACM, March 1982.

[58] Jonathan A. Rees. *The T Manual.* Computer Science Department, Yale University, New Haven, Connecticut, fourth edition, January 1984.

[59] Jonathan A. Rees and Norman I. Adams. T: A dialect of Lisp or, Lambda: The ultimate software tool. In *Conference on Lisp and Functional Programming*, pages 114–122, ACM, 1982.

[60] Jonathan A. Rees and William Clinger. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12), December 1986.

[61] Mark B. Reinhold. *Typechecking Is Undecidable When 'Type' Is a Type.* Technical Report 458, MIT Laboratory for Computer Science, Cambridge, Massachusetts, December 1989.

[62] Paul Rovner. *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language.* Technical Report CSL-84-7, Xerox Palo Alto Research Center, July 1985.

[63] A. Dain Samples and Paul N. Hilfinger. *Code Reorganization for Instruction Caches.* Technical Report 88/447, Computer Science Division, University of California at Berkeley, October 1988.

[64] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. In *Functional Programming Languages and Computer Architecture*, 1993. To appear.

[65] Thomas D. Simon. *Optimization of an $O(N)$ Algorithm for N-body Simulations.* Bachelor's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, December 1991.

[66] Patrick G. Sobalvarro. *A Lifetime-based Garbage Collector for LISP Systems on General-Purpose Computers.* Bachelor's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1988.

[67] James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, May 1984.

[68] Guy L. Steele Jr. Fast arithmetic in MacLISP. In *Proceedings of the MACSYMA Users' Conference*, pages 215–224, 1977.

[69] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR Lisp architecture. In *International Symposium on Computer Architecture*, pages 444–452, IEEE, 1986.

[70] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, Connecticut, 1983.

[71] Edward R. Tufte. *Envisioning Information.* Graphics Press, Cheshire, Connecticut, 1990.

[72] David Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* Ph.D. thesis, Computer Science Division, University of California at Berkeley, February 1986. Also available as an ACM Distinguished Dissertation from the MIT Press, Cambridge, MA.

[73] Philip L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[74] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. *Caching Considerations for Generational Garbage Collection: A Case for Large and Set-Associative Caches.* Technical Report UIC-EECS-90-5, Software Systems Laboratory, University of Illinois at Chicago, Chicago, IL, December 1990.

[75] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *Conference on Lisp and Functional Programming*, pages 32–42, ACM, 1992. An earlier version appeared as [74].

[76] Feng Zhao. *An $O(N)$ Algorithm for Three-dimensional N-body simulations.* Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, October 1987.

[77] Benjamin Zorn and Dirk Grunwald. *Empirical Measurements of Six Allocation-Intensive C Programs.* Technical Report CU-CS-604-92, Department of Computer Science, University of Colorado at Boulder, July 1992.

[78] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* Ph.D. thesis, Computer Science Division, University of California at Berkeley, December 1989. Available as UCB/CSD Technical Report 89/544.

[79] Benjamin G. Zorn. *The Effect of Garbage Collection on Cache Performance.* Technical Report CU-CS-528-91, Department of Computer Science, University of Colorado at Boulder, May 1991.