Cache Performance of Garbage-Collected Programs

Mark B. Reinhold
NEC Research Institute
Four Independence Way
Princeton, New Jersey 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 programs either argues or assumes 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 paper 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. Garbage-collected programs written in a mostly-functional style should perform well when simple linear storage allocation and an infrequently-run generational compacting collector are employed; sophisticated collectors intended to improve cache performance are unlikely to be necessary. As locality becomes ever more important to program performance, programs of this kind may turn out to have a significant performance advantage over programs written in traditional languages.

1. Introduction

One of the most prominent 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 [13, 14]. 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 present 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.

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 heap-allocated data objects so that most working data is kept in physical memory [7, 12, 25, 35]. Recently, several researchers have suggested that a collector might also be used to improve cache performance, by designing it to move objects so that most working data is kept in the cache [39, 42]. Because caches are so much smaller than main memory, a collector of this kind must be invoked more frequently than ordinary collectors if it is 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.

The proponents of aggressive collection either argue or assume that programs written in garbage-collected languages will have poor cache performance if little or no garbage collection is done. In contrast, the primary claim of this paper is that many such programs are naturally well-suited to the direct-mapped caches typically found in high-performance computer systems. This conclusion, and its corollaries, rest upon a study of five nontrivial, long-running Scheme programs, compiled and run in a high-quality Scheme system.

After a survey of prior work in §2, a brief description of the test programs in §3, and a delineation of the cache design space in §4, the results of two cache-performance experiments are presented. A control experiment, described in §5, shows that the test programs have good cache performance without any garbage collection at all. A second experiment, discussed in §6, shows that the programs should perform well with a simple, infrequently-run generational compacting collector.

The goal of the remainder of the paper is to generalize these results to other Scheme programs and to programs in other garbage-collected languages. In §7, 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. The analysis is generalized in §8, where 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. Therefore the conclusion that a simple, infrequently-run generational compacting collector should lead to good program performance is widely applicable, although it may not apply to programs with tight constraints on memory usage or response time. The paper closes by conjecturing that garbage-collected languages may have a significant performance advantage over more traditional languages on fast computer systems.

To appear in *Conference on Programming Language Design and Implementation*, June 1994. Copyright © 1994, ACM.

2. Prior work

Generational collectors can improve the performance of virtual memories by repeatedly reusing a modest amount of memory on a relatively small time scale. Wilson, Lam, and Moher were the first to suggest that this idea could be applied on an even smaller scale to improve the performance of caches [38, 39]. While intuitively appealing, it is not obvious that this approach will work once the fundamental differences between virtual memories and caches are taken into account. In particular, the cost of a page fault is many orders of magnitude larger than that of a cache miss, memory pages are several orders of magnitude larger than cache blocks, and virtual memories typically employ some approximation to 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 is here called 'aggressive garbage collection.' An aggressive garbage collector is essentially a generational collector [2, 22, 36] with a new-object area, or first generation, that is sufficiently small to fit mostly or entirely in the cache. From their measurements of four Scheme programs, Wilson *et al.* conclude that an aggressive collector can yield lower data-cache miss ratios than a Cheney-style compacting semispace collector [6].

Along similar lines, Zorn compared two different generational collectors. One, a noncompacting mark-and-sweep collector, moves objects only when they are advanced from one generation to the next; the other is a more traditional copying collector [41, 42]. Zorn measured the cache performance of four large Lisp programs running with these collectors in various configurations, ranging from frequently-run to very aggressive. He shows that the data-cache miss ratios of the programs are improved by the collectors, and conjectures that aggressive collection will be an effective means for improving program performance.

The arguments made in favor of aggressive garbage collection share three flaws. First, the supporting measurements use data-cache miss ratios as the primary metric of program performance, but miss ratios are not necessarily correlated with program running times; specifically, they do not account for the temporal costs of cache activity and garbage collection. Second, the measured program runs make, at most, only a few tens of millions of references, but with large caches it is especially important to use long program runs in order to obtain believable results from trace-driven cache simulations [4]. Finally, and most importantly, the proponents of aggressive collection either argue or assume that programs written in garbage-collected languages will have poor cache performance if little or no garbage collection is done. They do not, however, provide measurements to support this conclusion; in particular, they do not measure the performance of their test programs when run with a simple, infrequently-run garbage collector or with no collector at all. Without such control experiments, it is impossible to argue that a sophisticated collection strategy is desirable, much less necessary.

Diwan, Tarditi, and Moss recently published results supporting the proposition that garbage-collected languages can have good cache performance [8, 9]. Using a temporal metric, rather than miss ratios, they studied the cache performance of eight fairly substantial ML programs running in Standard ML of New Jersey [3, 24]. While they used a more accurate cache and memory-system simulator than that employed here, they only considered configurations similar to those of currently-available machines, and they did not measure cache and collector costs separately.

3. Test programs

The test programs studied in this paper are written in Scheme, a lexically-scoped dialect of Lisp that supports first-class procedures [31]. 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 ORBIT, one of the best Scheme compilers currently available [19, 20, 29, 30]. Having been in production use for several years, T has been used by many people to write nontrivial programs.

The five test programs and their input data are: ORBIT, the native compiler of the T system, compiling itself; IMPS, an interactive theorem prover [10], running its internal consistency checks and proving a simple combinatorial identity; LP, a reduction engine for a typed λ-calculus [1, 32], 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 [34, 40], computing the accelerations of 256 point-masses distributed uniformly in a cube and starting at rest; and GAMBIT, another Scheme compiler [11], quite different from ORBIT, compiling the machine-independent portion of itself.

These programs represent several different kinds of applications and programming styles. They vary in size, but each allocates many megabytes of data and executes billions of instructions:

	Lines	Alloc	Insns	Refs
ORBIT	15,332	94.4MB	3.68E9	1.03E9
IMPS	42,119	41.1MB	4.13E9	1.09E9
LP	2,981	58.6MB	2.21E9	.64E9
NBODY	857	126.1MB	2.43E9	.63E9
GAMBIT	15,004	106.9MB	7.35E9	2.00E9

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 loads and stores made by each program when run, without garbage collection, on its input data. These program runs are significantly longer than those used in most previous studies of the cache performance of garbage-collected languages [39, 42].

The programs were compiled and run in version 3.1 of the T system running on a MIPS R3000-based computer [17]. Measurements of memory and cache behavior were made by running each program, together with the T system itself, under an

instruction-level emulator for the MIPS architecture; neither the programs nor T were modified in any significant way.

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 left to future work.

4. Cache design parameters

The portion of the cache design space considered in this paper is constrained 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 [15, 27]; they are the most common type of cache in current high-performance computers. The caches are assumed to be virtually, rather than physically, indexed.

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

The caches are assumed to have a write-miss policy of write-validate; this is equivalent to write-allocate with sub-block placement, with a sub-block size of one word. In contrast to the more common fetch-on-write policy, write-validate avoids fetching the contents of memory blocks in which every word is written before being read. Jouppi has demonstrated that write-validate can yield significant performance improvements for C and Fortran programs [16]; Koopman *et al.* first noted the benefits of this policy for garbage-collected programs [18]. The impact of fetch-on-write upon the performance of the test programs will be discussed briefly in §5.

The temporal cost of writing data to main memory, which depends upon the write hit policy, is not analyzed in detail. Properties of practical memory systems and of the test programs themselves imply that these costs should be small [33]; preliminary measurements support this conclusion.

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

5. Cache performance without garbage collection

In order to determine the extent to which the cache performance of the test programs can be improved, the control experiment measures their cache performance when they are run without any garbage collection at all. This is done simply by disabling the collector; during each program run, data objects are allocated linearly in a single contiguous area. If these measurements 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 programs have good cache performance. The results will be described in terms of cache overheads, which express the temporal cost of cache activity relative to the programs' idealized running times.

The time required to service a 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. For concreteness, the miss penalties used here are based upon the main-memory system studied by Przybylski [27, §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.

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	

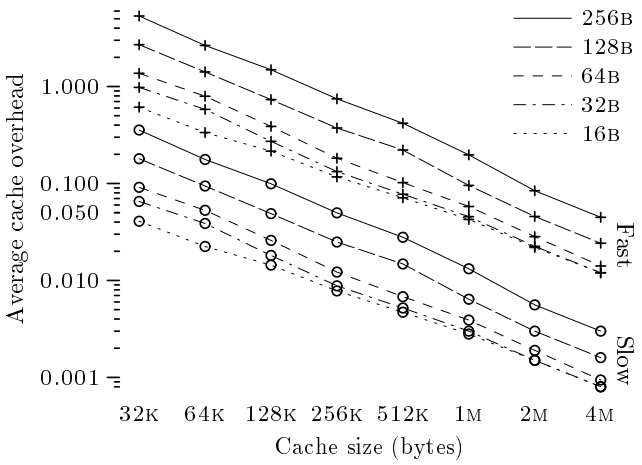
In light of expected improvements in memory-chip technology, especially improvements in bandwidth, the fast-processor penalties are conservative [28]. The hit time, *i.e.*, the time required to access a block that is already in the cache, is assumed to be one cycle for both processors. Thus, if a reference hits in the cache, the processor does not stall.

The *cache overhead* of a program, O_{cache}, is the amount of time spent waiting for misses to be serviced 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 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* [4] is one plus the overhead of each cache in the memory hierarchy.

With these assumptions and definitions in hand, the average 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 (O_{cache}), across all programs, for the given block size, cache size, and processor speed. The test programs' individual cache overheads are all close to the average.

For the slow processor, even a small 32KB cache has an 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 are likely to 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, larger caches and smaller block sizes always yield superior performance.

The cache overheads shown above assume a write-miss policy of write-validate. For the Scheme test programs, write-validate always outperforms the more common fetch-on-write policy, but sometimes by only a small margin.* For a given program, the number of fetches avoided by write-validate depends inversely upon the block size and is independent of the cache size. For the slow processor, a fetch-on-write policy never increases the average cache overhead by much more than one percent. For the fast processor, the increased overhead due to fetch-on-write is more significant, ranging from less than four percent, for 256-byte blocks, to nearly 20%, for 16-byte blocks. With the cache sizes considered here, it is impossible to attain an overall cache overhead of less than five percent for the fast processor without a write-validate policy; an overhead of less than ten percent, however, is achievable with a 2MB cache and 128-byte blocks.

The above overheads do not include the cost of writing data from the processor to memory, in the case of a write-through cache, or from the cache back to memory, in the case of a write-back cache. Preliminary measurements have shown that the write overheads of write-back caches are low. For the slow processor, write overheads are almost always less than one percent; for the fast processor, write overheads for caches of 1MB or more are almost always less than three percent. The write overheads of write-through caches have not yet been measured, and may be somewhat higher.

The control experiment has revealed, then, that the test programs have good cache performance when run without any garbage collection at all. With appropriate write-miss and write-hit policies, it is possible to achieve a cache overhead of less than five percent. Overheads are higher with less favorable policies; for the fast processor, they approach thirteen percent assuming realistic cache configurations. With so little room for improvement, it is not clear that improving cache performance should be a priority. No method for improving cache performance that imposes a significant overhead of its own could be effective.

6. Program performance with a simple collector

In practice, it is not possible to run programs with an unbounded amount of physical memory, so some garbage collection must be done in order to ensure good virtual-memory performance. Because a single page fault may cost hundreds of thousands, if not millions, of processor cycles, spending some computational effort in a collector in order to avoid paging is worthwhile. The results of the control experiment suggest that a good collector will be one that collects rarely, in order to approximate the non-collection case and thereby take advantage of the programs' naturally good cache performance, yet frequently enough to minimize 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 Cheney-style compacting semispace collector [6]. Generational compacting collectors typically provide better performance than Cheney's algorithm, but the T system does not include such a collector. All but one of the programs perform well with the Cheney collector, and should therefore perform well with a simple and infrequently-run generational compacting collector; the remaining program requires such a collector for good performance.

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 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

*In contrast, the ML programs studied by Diwan *et al.* appear to require write-validate for good cache performance [9]. This is most likely due to their extremely high allocation rates, which are, in part, a result of the fact that SML/NJ allocates procedure-call frames in the heap rather than on a stack.

*Cf. White's proposal for improving virtual-memory performance [37].

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. For the test programs, the cost of rehashing is usually small.

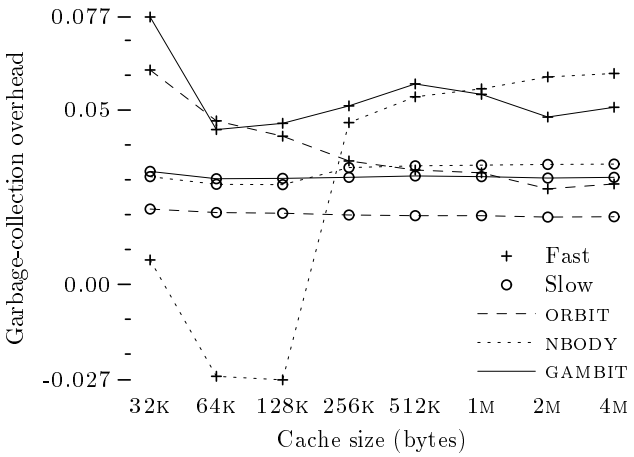
When run with a given collector, the *garbage-collection overhead* of a program is the sum of these temporal costs, expressed as a fraction of the program's idealized running time. That is,

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

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 running time of a program, taking both cache and collection costs into account, is $(O_{cache} + O_{gc} + 1) \times I_{prog}$. Because O_{gc} can be negative, it is possible for the sum of the overheads 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.

Program performance. When run with the Cheney collector, configured to use 16MB semispaces, the 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 hypothetical processor; 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 affect a program's cache performance by improving or degrading its reference locality as it moves data objects in memory, so another source of variation is the extent to which this type of effect 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 non-garbage 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 *collide*, because they share the same cache block, and they are referenced in such a way that they frequently displace each other. 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. To eliminate cache thrashing does not require a specialized garbage collector, but can be achieved by straightforward static methods that move frequently-accessed objects so that they do not collide [33].

The above 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 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, for it would copy the live objects in younger generations more frequently than those in older

*When the collector runs, the cache simulator employs a fetch-on-write policy; thus this graph slightly over-reports collection overheads.

generations. Although such a collector would impose costs beyond those of the Cheney collector, including the overheads of managing several generations and of detecting and updating pointers from old objects to new objects, 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. With a smaller first generation, an aggressive collector will be invoked more frequently than a non-aggressive generational collector. An aggressive collector will incur all the costs of an ordinary generational collector; moreover, because more frequent collections leave less time for new objects to become garbage before being copied to the next generation, an aggressive collector will spend relatively more time copying objects from the new-object area. 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.

7. Analysis of memory behavior

The control experiment showed that the five Scheme test programs have good cache performance when run without a garbage collector or any other mechanism that might improve reference locality; the second experiment showed that the programs should perform well with an infrequently-run generational compacting collector. The analysis presented in this section 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 the next section, the analysis will be generalized to other Scheme programs and to programs in other garbage-collected languages, thereby generalizing the experimental conclusions.

Some foundations must be laid before proceeding with the analysis. 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, and a coarse-grained unit of time will be defined.

The analysis will then show that linear allocation spreads dynamically-allocated data objects both spatially, throughout the cache, and temporally, within each cache block. Most dynamic objects are so short-lived that they are *dead* (*i.e.*, will not be referenced again) by the time their cache blocks are re-used for newer dynamic objects; thus, assuming no other interference, dynamic objects will be allocated, live, and die entirely in the cache. Nearly all other objects are not very active and are not referenced many times, so references to them cannot be a significant source of interference. The programs do contain some long-lived and frequently-referenced objects; these objects are very rare, however, and turn out to improve cache performance more often than they degrade it.

Sweeping the cache. Practical memory systems are designed to move fixed-size blocks between main memory and the cache. Thus the behaviors of blocks, rather than those of data objects, are the focus of the analysis. Block behaviors, of course, reflect the behaviors of the objects that they contain. Most Scheme objects are just a few words long, so a block typically contains at least a handful of objects.

When the garbage collector is disabled, the memory blocks for dynamically-allocated data objects (*i.e.*, *dynamic blocks*) are contained in a single contiguous area. The *allocation pointer* contains the address of the next available dynamic word and is incremented by each allocation action; with the collector disabled, it starts at the base of the dynamic area and grows upward, without bound, until the end of the run.

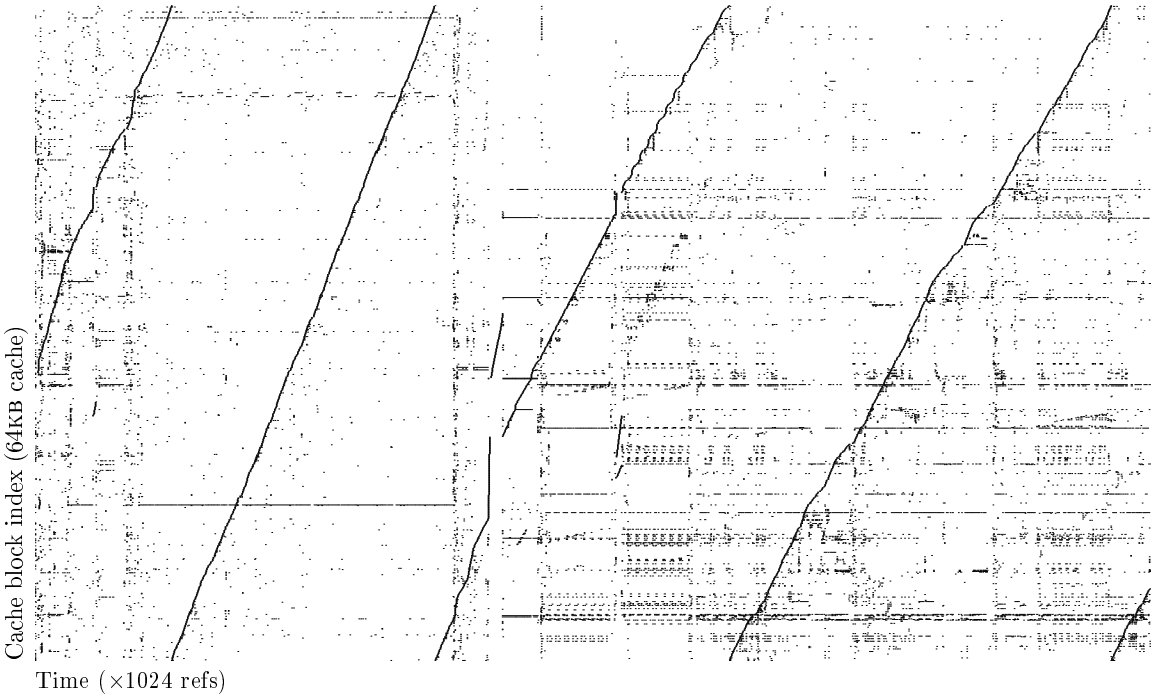
Each time an object is allocated, its component words are initialized, typically in ascending address order. Each time an initialization store reaches a new dynamic memory block, an *allocation miss* occurs, flushing the cache block, if necessary, and assigning it to the new memory block. A direct-mapped cache maps a memory block to a cache block by taking the memory block's index modulo the cache size (in blocks); thus the allocation pointer continually sweeps the cache from one end to the other, leaving a trail of newly-allocated objects.*

Because the advancement of the allocation pointer entails cache misses, each sweep through the cache appears as a broken diagonal line when cache misses are plotted as a function of time. Shown on the next page is a partial cache-miss plot for a short run of ORBIT in a 64KB, direct-mapped cache with 64-byte blocks. The horizontal x axis 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 in the 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.

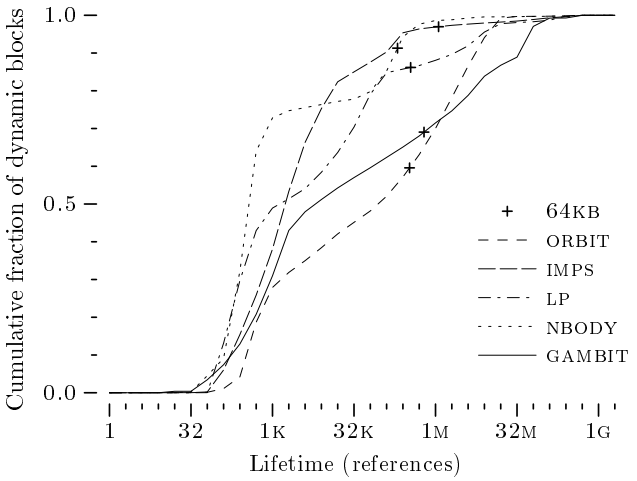
The slope of an allocation-miss line at a given point in time reflects the program's allocation rate, relative to its reference rate, at that time. The partial run shown in the plot is typical in that the allocation rate usually changes slowly, but sometimes changes quickly. For example, the nearly vertical segment in the lower part of the second full line 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.

The allocation pointer must traverse the entire cache before it revisits a cache block, so some time will elapse before a cache block is flushed and reassigned to another memory block due to allocation activity. In any given cache block, each interval between two successive allocation misses is an *allocation cycle*. 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. For the test programs, allocation cycles in 64KB caches range from several hundred thousand to two million references in length.

*With a fetch-on-write policy, each allocation miss causes the contents of the new block to be fetched from memory. Each word in a dynamic block is initialized before it is read, so the data retrieved by this fetch will never be used. A write-validate policy avoids these useless fetches.



One-cycle dynamic blocks. The *lifetime* of a memory block is the closed time interval bounded by its first and final references. A natural consequence of a mostly-functional programming style is that most dynamic blocks have extremely short lifetimes:



This graph shows the cumulative frequency distributions of dynamic-block lifetimes in each test program, for a block size of 64 bytes. A point on a curve shows, in its y value, the fraction of dynamic blocks with lifetimes no greater than its x value. In two of the programs, about half of all dynamic blocks have lifetimes no greater than 64K references; in the other three programs, this is true of even more blocks.

Most, and sometimes nearly all, dynamic blocks have such short lifetimes that they are *one-cycle* blocks. A one-cycle block is a block whose lifetime is completely contained in its initial allocation cycle. Because every one-cycle block is dead by the time the allocation pointer revisits its cache block, one-cycle blocks cannot interfere with each other. Absent other interference, one-cycle blocks will be allocated, live, and die entirely in the cache.

Even when the test programs are run in a relatively small 64KB cache, most dynamic blocks are one-cycle blocks. In the above graph, the fraction of dynamic blocks that are one-cycle blocks in a 64KB cache is shown by the marker on each curve. In all of the programs, at least half, and often more than eighty percent, of all dynamic blocks are one-cycle blocks.

Interference. If all memory blocks were one-cycle dynamic blocks, then the only cache misses would be allocation misses. While most, and sometimes nearly all, memory blocks in the test programs fall into this category, a significant number do not. Some dynamic blocks are *multi-cycle* dynamic blocks; *i.e.*, they survive their initial allocation cycles. There are also *static* blocks, which exist when a program starts running. Static blocks contain the program itself, the procedure-call stack, and data structures and code for the compiler, library, and runtime system. The remainder of the analysis argues that, whatever their category, most memory blocks are referenced in such a way that they are not significant sources of cache interference.

Interference from dynamic blocks. Most dynamic blocks have extremely short lifetimes, and so they have little opportunity to cause other blocks to be removed from the cache; thus they cannot be significant sources of interference.

Multi-cycle dynamic blocks have longer lifetimes, but they are unlikely to be significant sources of interference. A memory block is *active* in an allocation cycle if it is referenced at least once in that cycle. Most multi-cycle blocks are not very active: When each test program is run in a 64KB cache, at least 90% of its multi-cycle blocks are active in no more than four distinct allocation cycles. Multi-cycle blocks are likely to be distributed throughout the cache, since it is unlikely that

the allocation times of multi-cycle blocks will be synchronized with the sweep of the allocation pointer through the cache.

The amount of interference created by all dynamic blocks is limited by the fact that most such blocks are referenced just a few times, regardless of their lifetimes. If a block is not referenced many times, then it cannot be a significant source of interference. In the test programs, for example, with 64-byte blocks, most dynamic blocks are referenced between 32 and 63 times. A 64-byte block contains 16 words, so, on average, each word is referenced between two and four times. This figure is another consequence of the mostly-functional style typically used in Scheme programs. Because programmers are encouraged to create and use data structures freely, it is quite common, *e.g.*, for a list to be created by one procedure, passed to another, and then traversed just once or twice before being discarded.

Interference from static blocks. Static blocks are arranged in an essentially random fashion, so they are uniformly spread throughout the cache; one static block is about as likely as any other to collide with some other block. Nearly all static blocks behave in a manner similar to that of multi-cycle dynamic blocks: The amount of interference they can create is limited because they are only active in a few allocation cycles and they are not referenced many times.

A few static blocks are referenced far more frequently than all other blocks. Say that a block is *busy* if it accounts for at least one thousandth of a program's references. In the test programs, nearly all frequently-referenced blocks meet this threshold. In each program there are between 59 and 155 busy static blocks (*i.e.*, less than .02% of all active blocks), yet together they account, on average, for 75% of all references.

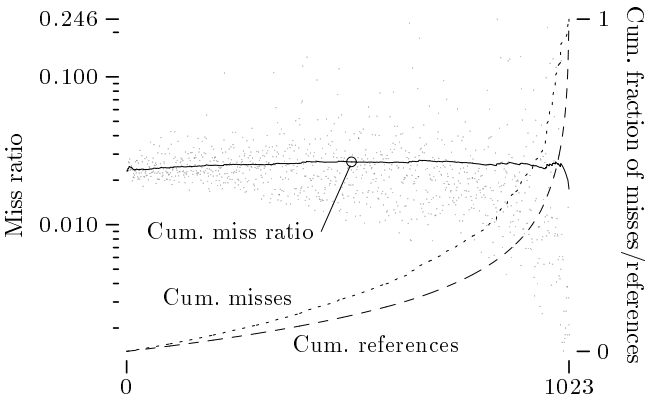
In the test programs, busy static blocks arise in three ways. Most commonly, they contain closures for frequently-called procedures. Busy blocks also contain the procedure-call stack; in each program, nearly all stack references are concentrated in a small, contiguous group of extremely busy blocks. Finally, the very busiest blocks contain a small vector internal to the T runtime system that accounts, on average, for 6.7% of all references.

From behavior to performance. 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 and a few non-busy static blocks; these blocks are likely to be active in just a few 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. In the 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. In the 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. Since most cache blocks will not have any busy blocks mapped to them, their local performance will fall between these best and worst cases. 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.

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. Most cache blocks, with no busy blocks and 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 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 cache with 64-byte blocks 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, show the cumulative distributions among cache blocks of misses and references, respectively. A point on the dotted curve indicates, in its y value, the fraction of all misses (excluding allocation misses) that occur in the x^{th} least-referenced cache block or to cache blocks referenced no more than that block; similarly, the dashed curve accumulates cache-block reference counts. These 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 miss-ratio scale, which is logarithmic. 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 represent bad local perfor-

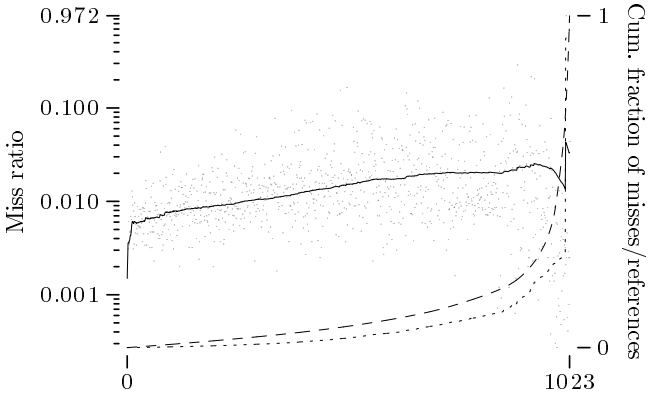
mance, while those in the lower quarter represent good local performance. Some of the less-referenced cache blocks perform badly, but the local miss ratios of most of these blocks fall into the central half of the graph. The hundred or so most-referenced cache blocks have local miss ratios ranging from very bad (the worst case) to very good (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 reflects, 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, shows 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 appears small, but in fact it falls from 0.027 to 0.017, a factor of about 1.6.

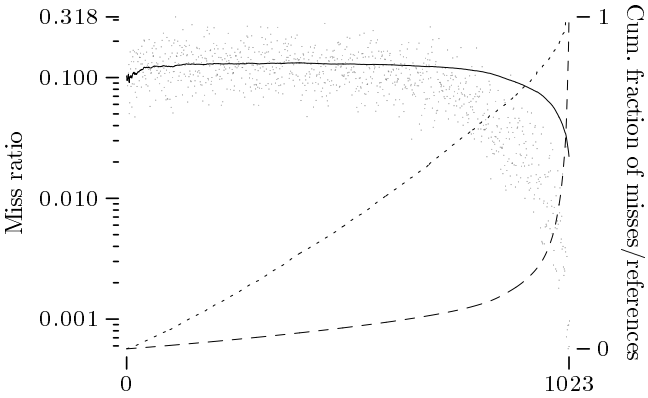
A similar pattern holds, with variations, for the other test programs when run in a 64KB cache. For example, NBODY exhibits a more pronounced concentration of references and misses in the last few cache blocks. In some programs, such as LP, there are no worst-case cache blocks, so the best-case blocks just improve the cache's global performance over that of the less-referenced cache blocks.

If a program thrashes, there will be jumps in the cumulative miss-ratio curve; IMPS, for example, thrashes in a 64KB cache:



The single jump implies that only one cache block is thrashing; it turns out that this cache block sees references to a busy stack block and a busy non-stack static block in almost perfect alternation. As noted in §6, to avoid thrashing does not require a special-purpose garbage collector, but merely some care in choosing the locations of busy objects.

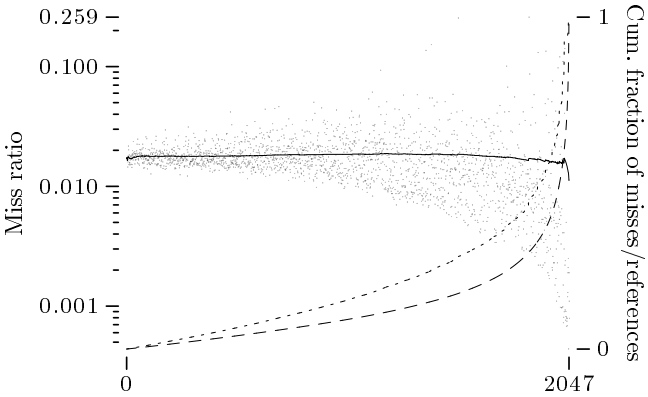
Sometimes, misses are spread throughout the cache; this is the case for GAMBIT:



Most of the less-referenced blocks in GAMBIT perform badly, with typical local miss ratios roughly an order of magnitude higher than those seen in the other programs. This may be due to the fact that GAMBIT has many long-lived dynamic blocks; if these blocks are referenced around the same time, they could cause many cache misses. In the end, however, the best-case cache blocks again pull the global miss ratio down to a more satisfactory level.

The examples presented so far have been limited to 64KB caches. As was seen in §5, 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 are illustrated in the cache-activity graph of ORBIT running in a 128KB cache:



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.

The analysis has only examined 64-byte blocks. Most data 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 example, a larger fraction of dynamic blocks should be one-cycle blocks, and there should be relatively fewer busy blocks. Similarly, performing the measurements at a larger block size should reveal weaker extremes. These expectations are consistent with the observation, made in §5, that smaller block sizes yield superior performance, at least within the range of block sizes being considered.

8. Generalizing the analysis

According to the foregoing analysis, 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, one-cycle dynamic blocks;

(3) Nearly all other blocks are only active in a few allocation cycles, and are 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 1. 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 2. Properties (1)–(3) above hold across a range of programming styles in garbage-collected languages.

This conjecture is based upon the 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 they contain. In a language that encourages a more functional style than Scheme, *e.g.*, ML [26], 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 language that encourages a more imperative style than Scheme, *e.g.*, CLU or Modula-3 [5, 23], object lifetimes are longer, but allocation rates are lower. Therefore programs in these languages 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. A more imperative style may engender 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 §7 might not be seen in CLU or Modula-3 programs. Nonetheless, properties (1)–(3) may hold in languages that encourage a slightly more imperative style than Scheme.

These considerations lead, finally, to:

Conjecture 3. Allocation can be faster than mutation.

That is, on machines where cache performance can have a significant impact on 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 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 [21]. 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. Such languages may also have a significant performance advantage on machines where cache performance is important to program performance.

9. 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 either argues or assumes that programs written in these languages will have poor cache performance if little or no garbage collection is done. This paper 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.

From this conclusion it follows that the experimental results obtained for the five Scheme test programs should apply to a wide range of programs in a variety of garbage-collected languages. The best memory-allocation strategy will be linear allocation, which distributes dynamic objects both spatially, throughout the cache, and temporally, within each cache block. The best garbage-collection strategy will be one of infrequent generational compacting collection, which can approximate the idealized case of no collection and thereby takes advantage of a program's naturally good cache performance.

This advice is not universally applicable, for infrequent collection is not appropriate for all programs. If a program must run in a small physical memory, then the space efficiency of frequent generational collection or of mark-and-sweep methods may be preferred. For an interactive program, guaranteeing a bound on collector pause times may be more important than optimizing the program's running time. When infrequent collection is acceptable, however, complex and costly means for improving cache performance, such as aggressive collection, are unlikely to be necessary.

The measurements and analysis presented here also provide guidance for the future, when processors will be faster, relative to main memories, than they are now. Mostly-functional, garbage-collected programs will benefit from some means for avoiding useless memory fetches, such as a write-miss policy of write-validate. Because such programs naturally make good use of direct-mapped caches, they may prove to have a significant performance advantage over programs written in traditional languages.

Acknowledgements

This work owes much to the enthusiasm, encouragement, and advice of John Guttag. Helpful comments on drafts of this material were provided by Mark Day, Bert Halstead, Suresh Jagannathan, Butler Lampson, Scott Nettles, Jim O'Toole, Tim Shepard, Guy Steele, and several anonymous reviewers. The authors of various software systems provided both code and assistance: Josh Guttman (IMPS), Mark Hill (the TYCHO cache simulator, which was used to validate the simulator employed here), David Kranz (ORBIT), and Tom Simon (NBODY).

Most of this work was performed at the Laboratory for Computer Science of the Massachusetts Institute of Technology, and 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.

References

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

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

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

[4] 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, California, September 1989.

[5] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. *Modula-3 Report (revised).* Research Report 52, Digital Equipment Corporation Systems Research Center, Palo Alto, California, November 1989.

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

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

[8] Amer Diwan, David Tarditi, and Eliot Moss. *Memory Subsystem Performance of Programs with Intensive Heap Allocation.* Technical Report 93-227, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1993.

[9] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with copying garbage collection. In *Symposium on Principles of Programming Languages*, pages 1–13, ACM, January 1994.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

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

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

[28] Steven A. Przybylski. DRAMs for new memory subsystems. *Microprocessor Report*, 15 February, 8 March, and 29 March 1993.

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

[30] 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.

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

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

[33] Mark B. Reinhold. *Cache Performance of Garbage-Collected Programming Languages*. Technical Report 581, MIT Laboratory for Computer Science, Cambridge, Massachusetts, September 1993.

[34] 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.

[35] 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, September 1988.

[36] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Symposium on Practical Software Development Environments*, pages 157–167, ACM, April 1984.

[37] Jon L. White. Address/memory management for a gigantic LISP environment or, GC considered harmful. In *Lisp Conference*, pages 119–127, ACM, July 1980.

[38] 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.

[39] 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 [38].

[40] 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, October 1987.

[41] 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.

[42] 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.